**Szegedi Tudományegyetem**

**Informatikai Intézet**

**Szakdolgozat**

**Horváth Bendegúz**

**2022Szegedi Tudományegyetem**

**Informatikai Intézet**

**Ládapakolás túltöltéssel**

Készítette:

**Horváth Bendegúz**

Programtervező informatikus szakos hallgató

Témavezető:

**Dr. Balogh János**

egyetemi docens

**Szeged**

**2022**

Tartalomjegyzék

[1. Bevezetés 4](#_Toc99793088)

[2. Felhasznált algoritmusok 5](#_Toc99793089)

[2.1 A túltöltéses ládapakolás. 6](#_Toc99793090)

[2.2. First Fit 9](#_Toc99793091)

[2.3. Best Fit 11](#_Toc99793092)

[2.4. Harmonic/worst Fit 11](#_Toc99793093)

[3. Fejlesztői dokumentáció 12](#_Toc99793094)

[3.1. Fejlesztői környezet 12](#_Toc99793095)

[3.2. Flowchart 12](#_Toc99793096)

[3.2.1. First Fit algoritmus 12](#_Toc99793097)

[3.2.2. második algoritmus -> függvények 14](#_Toc99793098)

[3.2.3. harmadik algoritmus -> függvények 14](#_Toc99793099)

[3.3. A futtatáshoz szükséges bemeneti adatok – python könyvtárak 14](#_Toc99793100)

[3.5. Bővítési lehetőségek 14](#_Toc99793101)

[4. Forrásjegyzet 15](#_Toc99793102)

# 1. Bevezetés

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. (Az offline probléma esetén az összes elem előre adott, azaz ismerjük minden elem méretét.)

# 2. Felhasznált algoritmusok

A két leghíresebb online ládapakolási algoritmus 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 beelfér, akkor mindkét algoritmus nyit egy új ládát, és abba pakolja az érkező elemet.

Ezeknek az algoritmusoknak az elemzése egészen Ullman munkájáig nyúlik vissza **[17]**. Sgall [16] cikkében található meg az online ládapakolásról egy összefoglaló áttekintés.

Egy online algoritmus teljesítménye mérhető az aszimptotikus versenyképességi arány, valamint az abszolút versenyképességi arány alapján.

Az *aszimptotikus versenyképességi arány*t a következőképpen határozzuk meg:

ahol az optimális megoldás által használt ládák számát, míg az ALG algoritmus által használt ládák számát jelöli bármely σ bemenetre (inputra).

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 abszolút versenyképességi arányt a következőképpen határozzuk meg:

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

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.

A klasszikus feladaton kívül online ládapakolási feladat számos változatát vizsgáltak. 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). Az az úgynevezett online változó méretű ládapakolási probléma. Kinnerly és Langston [13] egy módosított FF-típusú algoritmust használt erre, a FF-et a felhasználó által meghatározott kitöltési tényezővel (FFf), 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). Az OOBP-ben 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). Ebben az egyes ládákban lévő tárgyak súlyának kisebbnek kell lennie 1-nél, ha a legkönnyebb elem (legkisebb méretű) eltávolítanánk. Valamint a másik az ún. lazy 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 ládát esetleg).

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 elemek egy halmaza adott, és a cél a használt tárolók költségének minimalizálása.

## 2.1 A túltöltéses ládapakolás.

***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 kell akkor felszámolnunk, ha egy processzorhoz rendelt feladatok által igényelt kapacitások összege meghaladja a kiszolgáló kapacitást.

Célunk az összes feladat kiszolgálásának 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.

Azaz 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, amely két részből tevődhet össze:

* 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.
* A láda kapacitásának túltöltése, vagyis a láda kapacitásának túllépése előírt költséggel jár. Azaz az előző költségen kívül fizetünk minden olyan ládáért, amely túlterhelt, túltöltött. Az -et meghaladó minden töltésért arányos költséget fizetünk. Azaz a túlterhelés költsége lineárisan függ a túlterhelés méretétől.

Megjegyezzük, hogy nem túltöltött ládáknál csak az első, fix költséget kell kifizetnünk, mint ládahasználati díjat, 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 az összköltségének minimalizálása (azaz a két költség összegét minden ládára összegezve).

Tehát a feladat egy konkrét példánya olyan, hogy kapunk egy elemből álló sorozatot, ahol nem tudjuk előre, hogy mekkora az n 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 (felhasznált) láda költsége ahol az ládában lévő elemek összmérete. A cél az, hogy pakoljunk minden egyes tárgyat ládába, úgy, hogy minimalizáljuk a felhasznált ládák költségeinek összegét. Ezt a problémát online ládapakolásnak nevezzük lineáris túlterhelési költséggel (online bin packing problem with linear overload cost, BPOC).

Ebben a dolgozatban a BPOC abszolút versenyképességi arányát vizsgáljuk.

Az [LS21] cikkben a szerzők a versenyképességi arány alsó 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 (azaz 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.

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 (azaz , ha *c* a fenti intervallumokba esik). Azaz ezzel felső korlátokat is megadtak a BPOC feladatra, bárhová is essen . ( pedig alsó korlát, bárhova is essen .)

## 2.2. First Fit

A First-Fit algoritmus minden újonnan érkező elemet az első olyan ládába pakol, amelybe befér vagy, ha nem fér be 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 méretében megengedünk 1-nél magasabb ládákat. 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.

De persze 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, azaz a túltöltés költsége.

## 2.3. Best Fit

## 2.4. Harmonic/worst Fit

# 3. Fejlesztői dokumentáció

## 3.1. Fejlesztői környezet

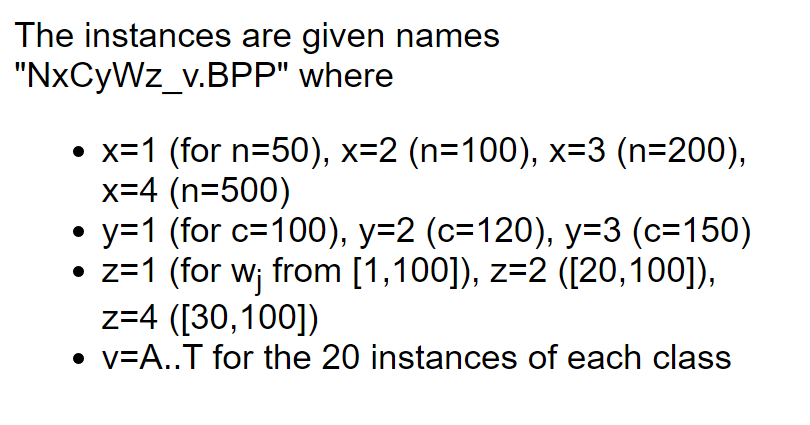
Az algoritmus implementációjához a Python programozási nyelvet választottam. A Python programozási nyelvet Guido van Rossuam holland programozó kezdte el fejleszteni 1989-ben, majd 1991-ben hozta nyilvánosságra. Választásom legfőbb szempontja a nyelv magasszintű adatstruktúrái, valamint a kevésbé komplex szintaxisa volt. Rövid programok írása gyors, egyszerű és jól áttekinthető. Ezenívül 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, ezt nem kell előre definiálni, ez nagyban megkönnyíti a nyelv használhatóságát.

Az általam írt kód Python 3.10.4 –ben készült, a program elkészítéséhez az OS és a SYS könyvtárakat használtam fel. Az előbbi különböző funkciókkal teszi lehetővé az operációs rendszerrel való kommunikálást. E könyvtár segítségével operációs rendszertől függően tudunnk műveleteket végrehajtani. Ilyenek például a fájlok és mappák tartalmának lekérdezése, törlés, létrehozás, átnevezés funkciók. Ezenkívül tudunk a mappaszerkezetben lépegetni, fájlok méretét meghatározni. Az utóbbi segítségével pedig a futási környezet különböző részeihez kapunk hozzáférést. Lehetővé teszi interpreterrel való műveletek végrehajtását, mivel hozzáférést biztosít az interpreterrel szoros kölcsönhatásban álló változókhoz és függvényekhez.

Fejlesztő környezetnek a JetBrains által forgalmazott PyCharm IDE-t használtam. Ez a környezet 2010-ben jelent meg, Java és Python nyelven lett megírva, a legtöbb operációs rendszeren elérhető. A terméknek van ingyenes verziója is, azonban egyetemi licenszel a professzionális verziót is tudtam használni. PyCharm segítségével a forráskódjainkat projektbe tudjuk szervezni, ezen belül külön-külön lehet őket futtatni. Az IDE feltérképezi a használt Python könyvtárakat, kódkiegészítő funkciókat, valamint a projekt egészére terjedő refaktorálást is lehetővé tesz.

A kódom verziókövetéséhez Git-et használtam, a kódomat egy publikus GitHub repository-ba töltöttem fel minden fejlesztés után. Ehhez a GitHub Desktop verzióját használtam, mivel a grafikus felület nagyban felgyorsítja a Git használatát.

Mivel a megírt algoritmusok megfelelő terszteléséhez több féle beemeneti fájlra volt szükségem, így külső forrásokra volt szükségem. Konzulensem tanácsára az M. Delorme, M. Iori és S. Martello BPPLIB ládapakolás probléma könyvtárban kerestem megfelelő input csomagot. A választás végül a SCHOLL 1 nevezetű könyvtárra esett, mivel ez egy sokszínű példatár, amely segítségével különböző inputokra tudtam futtatni az algoritmusom. Ezenkívül 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] interevallumok 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, az alábbi kép szemlélteti ezt:



*3.1 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, összesen 720db feladattal.

## 3.2. Flowchart

### 3.2.1. First Fit algoritmus

Az FFO algoritmus pszeudó-kódja: Jelölje az algoritmus futása során az aktuális ládaszámot (az algoritmus által addig megnyitott ládák számát egy adott ponton), továbbá jelölje az elemek összes méretét, mennyiségét a -edik ládában.

Jelölje továbbá az algoritmus végén a ládaszámot.

A -edik láda (sőt tulajdonképpen minden láda) kapacitása az FFO esetében , azaz ebből 1 a hagyományos ládaméret és még lehet a túltöltés mérete egy ládában (az egy fölötti, egyet meghaladó rész maximum ennyi lehet az algoritmus definíciójában).

**1. algoritmus. A First Fit algoritmus rögzített túltöltéssel (First-Fit Algorithm with Fixed Overload, FFO)**

**1** **Input:** A túltöltési költség.

**2** **Inicializálás:** Állítsuk be O(c) értéket a fenti definíció alapján. Legyen ,

**3** **for each** elem **do** (minden érkező i inputelem – i a következő eleme az inputnak -- esetén csináljuk a következőt )

**4** **for each** láda **do** (minden létező j láda esetén csináljuk a következőt)

**5**

Tehát itt számoljuk ki, hogy mennyi lenne a láda szintje, ha a méretű elemet beraknánk az aktuálisan éppen magasságú -edik ládába, akkor annak magassága , és az a kérdés, hogy az elem befér-e, azaz ez meghaladja-e a megengedett magasságát a ládának.

Ha ez a különbség negatív, akkor nem rakhatjuk bele, mert meghaladná a láda általunk megengedett kapacitását. Ha ez a különbség egy pozitív (vagy legalábbis nemnegatív) érték, akkor berakhatjuk, több ilyen esetén az első ilyenbe, a legkisebb ilyen j-hez tartozó ládába. Az lesz a minimum, a pozitív (nemnegatív) j ládák közüli legkisbb indexű

**6** **end for**

**7** **if**

**8** rakjuk i-t abba a j ládába, amelyre ;

Tehát itt darab ládánk van aktuálisan, kérdés, hogy belefér-e az aktuális i elem valamelyikbe. Ha igen, a legkisebb indexű olyanba rakjuk. Ennek a ládának a szintje megnő -vel ekkor.

**9 else do**

**10** ; ; rakjuk az elemet a indexű ládába;

tehát ekkor nyitunk egy új ládát, h számú ládánk volt, ez lesz a h+1., az első h számú ládába ha beraktuk volna, mindnek 1 + O(c) felett lett volna a mérete, nyitunk egy új ládát, egy ilyen 1 +O(c) kapacitással, és az egyetlen elem benne a méretű i elem lesz, ez lesz a láda nyitószintje.

**11 end if**

**12 end for**

**12** **Output:** k=h számú láda; ezek szintje minden esetén

A bemenet egy érték, ami a túltöltés értékét jelöli. Ezután inicializáljuk a kódban használt értékeket

* Itt a túltöltés értékét jelöli,
* az aktuálisan megnyitott ládák számát,
* a j-edik láda mindenkori aktuális szintjét (az addig a pontig belepakolt, benne lévő elemek méretének összegét),
* az aktuálisan érkező elem méretét.
* a láda max kapacitása (1) + a túltöltés értéke, azaz 1+O(c)

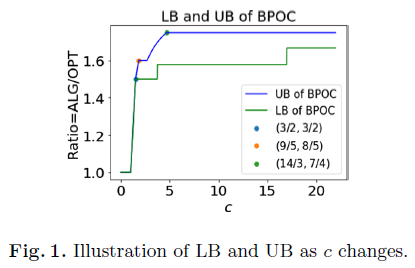
Ezután egy for each ciklussal végig megyünk az összes elemen egyesével. Ebbe a for ciklusba ágyazunk még egy belső for ciklust ami 1-től az összes (h számú) ládán végigmegy.

Minden lépésben meghatározzuk a dj értékét, am i azt jelenti, hogy ha az i-edik elemet a j-edik ládába raknnk be, akkor abban mennyi hely maradna. Ezt úgy számítjuk ki, hogy a láda általunk megengedett kapacitásából kivonjuk a láda eddig szintjének és az érkező elem pi méretének összegét.

A belső ciklusból kilépve egy if feltétellel ellenőrizzük, hogy létezik-e j, ami megfelel a {1,2…,h} halmaz egyik tagjának, vagyis van-e olyan j érték, amelyre 1 j h és dj 0, ha a feltétel teljesül az i értéket belepakoljuk a j-edik ládába, ahol min{j|dj }, majd a j-edik láda méretét súlyát (wj –t) megnöveljük pi-vel. Különben a h értékét megnöveljük eggyel, majd a h-adik új láda kapacitásának értékéül is az 1 + O(c)–t adjuk. Ezután az i elemee belerakjuk ebbe az újonnan nyitott h-adik ládába és a h-adik láda induló méreteként megadjuk a pi-t.

Miután lefutott a kód az output k = h darab láda lesz, ahol wj  az adott ládák mérete, ahol j = 1,2,…,k.

Az alábbi kép szemlélteti az FFO algoritmus alsó és felső korlátainak változását. Észrevehető, hogy bizonyos c értékeknél az alsó és felső korlátok egybeesnek.



Kép forrása: Luo, K. and Spieksma, F.C.R.: Online Bin Packing with Overload Cost, A. Mudgal and C. R. Subramanian (Eds.): CALDAM 2021, LNCS 12601, pp. 3–15, 2021. 6.oldal 1. ábra

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

### 3.2.2. második algoritmus -> függvények

### 3.2.3. harmadik algoritmus -> függvények

## 3.3. A futtatáshoz szükséges bemeneti adatok – python könyvtárak

## 3.5. Bővítési lehetőségek

# 4. Forrásjegyzet

**[LS21]** A fő hivatkozás:

Luo, K. and Spieksma, F.C.R.: Online Bin Packing with Overload Cost, A. Mudgal and C. R. Subramanian (Eds.): CALDAM 2021, LNCS 12601, pp. 3–15, 2021. DOI: https://doi.org/10.1007/978-3-030-67899-9\_1

**[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)

**[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

**[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)

**[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

**[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)

**[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

**[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)

**[13]** Változó ládaméretű ládapakolás:

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

**[5]** Variable Harmonic algoritmus:

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

**[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)

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

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

**[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)

**[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)

**[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)