**Universidad Peruana de Ciencias Aplicadas**

**Facultad de Ingeniería**



**Nombre del Proyecto:**

**“Empaquetamiento y Corte en 3D”**

**Curso: Complejidad Algorítmica**

**Profesora: Canaval Sanchez, Luis Martin**

**Alumnos:**

|  |  |
| --- | --- |
| Guillermo Gavilano Auris | 20171B294 |
| Diego Alonzo Hilario Callupe | 20171B079 |
| Ricardo Daniel Iglesias Espinoza | 20161C808 |

**Introducción**

El presente trabajo abordará el problema empaquetado tridimensional, que comúnmente presentan las empresas dedicadas a la logística, algunas tareas comunes donde se presentan estos problemas son al llenar contenedores, camiones, barco y/o aviones de carga, donde se desea hacer la menor cantidad de viajes posibles por ende deben asegurarse de enviar la mayor cantidad de elementos en cada transporte que realizan, dejando la menor cantidad de espacio. En la actualidad, existen diferentes heurísticas de las cuales se pueden hacer uso para implementar algoritmos que solucionen el problema.

En este proyecto se buscará y/o implementará tres algoritmos para poder darle solución a un problema real, mostrando su seudocódigo, algoritmo, complejidad, tablas comparativas del consumo de recursos de memoria y tiempo, y como conclusión elegir al mejor algoritmo basándonos en la complejidad, consumo de recursos y tiempo.

Índice

[**1.** **Objetivo del Estudiante (Student Outcome)** 4](#_Toc22847968)

[**2.** **Estado del Arte** 5](#_Toc22847969)

[**2.1 Algoritmo 1:** 5](#_Toc22847970)

[**Heurística Left-Bottom (BL)** 5](#_Toc22847971)

[**Heurística Finite-Next-Fit (FNF)** 5](#_Toc22847972)

[**2.2 Algoritmo 2:** 6](#_Toc22847973)

[**Heurística Next-Fit Decreasing Height (NFDH)** 6](#_Toc22847974)

[**2.3 Algoritmo 3:** 7](#_Toc22847975)

[**3.** **Aporte: Demuestra ética profesional** 8](#_Toc22847976)

[**4.** **Diseño de Aplicativo para Pruebas** 9](#_Toc22847977)

[**4.1 Pseudocódigo 1:** 9](#_Toc22847978)

[**4.2 Pseudocódigo 2: Heurística Next-Fit Decreasing Height (NFDH)** 9](#_Toc22847979)

[**4.3 Pseudocódigo 3:** 9](#_Toc22847980)

[**5.** **Validación de Resultados y Discusión** 10](#_Toc22847981)

[**6.** **Conclusiones y Trabajos Futuros** 11](#_Toc22847982)

[**7.** **Conclusiones** 12](#_Toc22847983)

[**8.** **Anexos** 13](#_Toc22847984)

[**9.** **Bibliografía** 14](#_Toc22847985)

# **Objetivo del Estudiante (Student Outcome)**

Nuestro objetivo como estudiantes del curso de Complejidad algorítmica y de la carrera de Ingeniería de Software es aplicar los conocimientos aprendidos hasta el día de hoy para solucionar y/o mejorar problemas de contexto real, permitiendo nuestro desarrollo en el reconocimiento de responsabilidades de manera profesional en situaciones de ingeniería. Además, buscamos desarrollar la ética y profesionalismo al citar y nombran los autores de los algoritmos y heurísticas de las cuales haremos uso para poder implementar algoritmos en 3D.

# **Estado del Arte**

## **2.1 Algoritmo 1 (AlgortGui):**

## **Heurística Left-Bottom (BL)**

Este algoritmo de empaquetamiento también sigue la heurística de Strip Packing. Consiste en ordenar todas las piezas a empaquetar de manera decreciente de acuerdo a su área, luego se empaqueta de forma que la siguiente pieza vaya lo más cerca posible de la parte inferior, y luego lo más cerca posible de la izquierda sin superponerse con ninguna otra pieza embalada. Hay que tener en cuenta que este algoritmo no rota las piezas a empaquetar simplemente las acomoda.

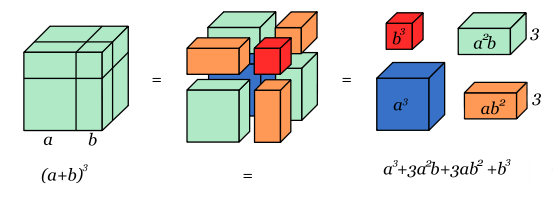
## **Heurística Finite-Next-Fit (FNF)**

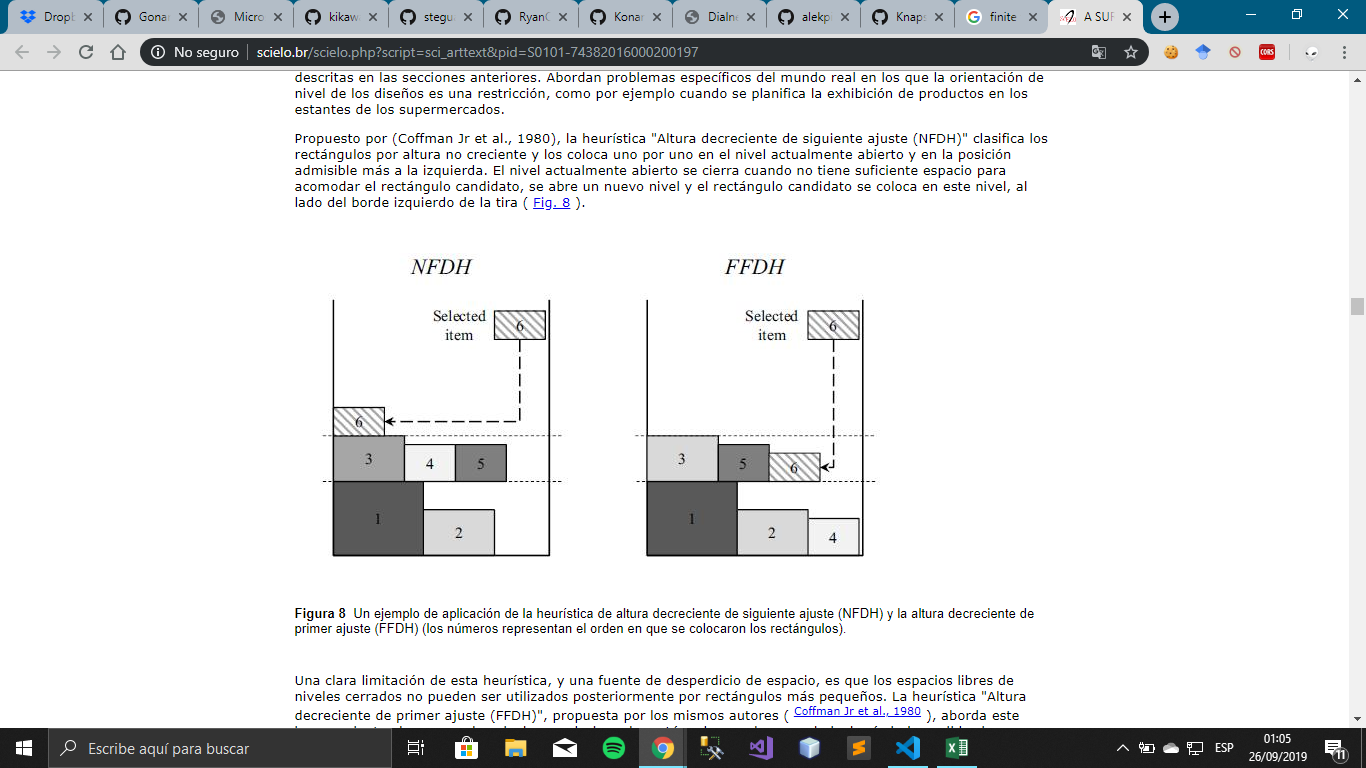
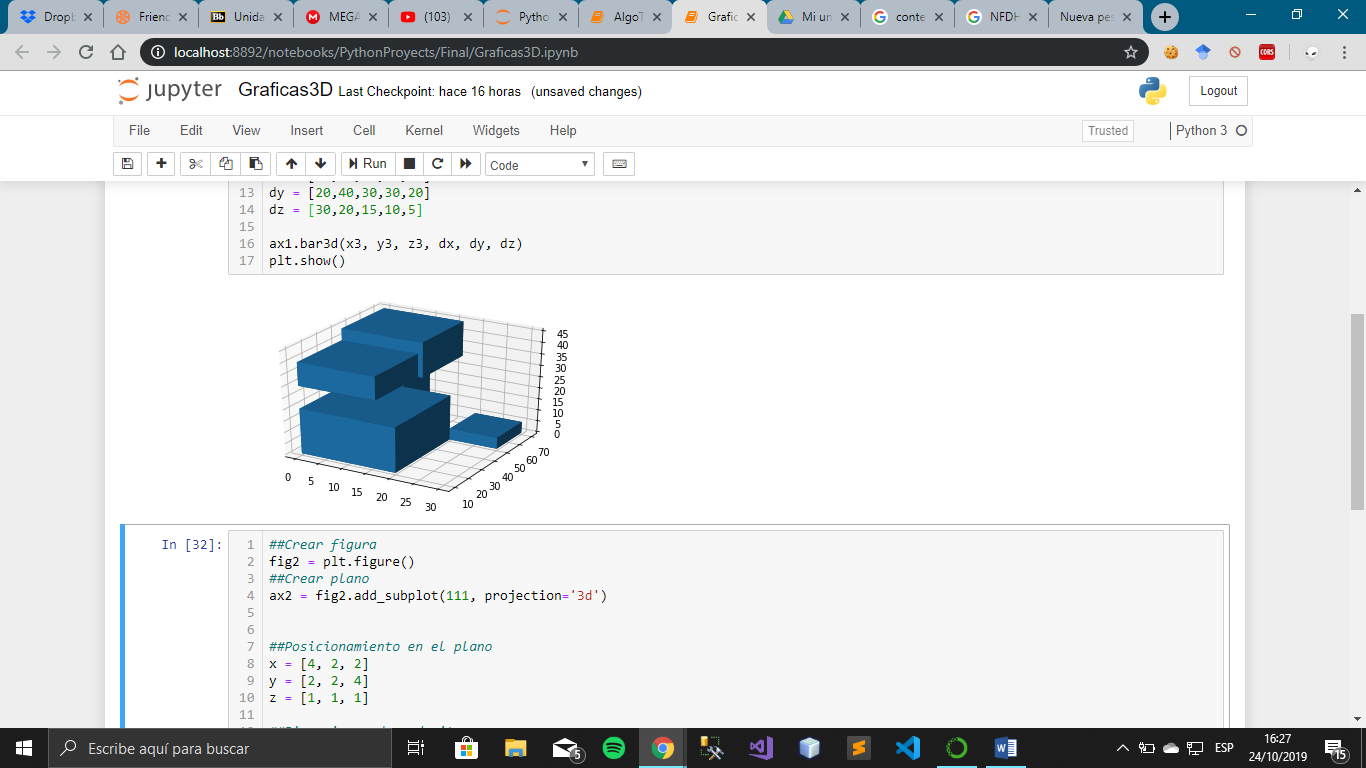
Inicialmente se tienen todos los contenedores vacíos. Se empieza con el primer el primer ítem (i=1). Si el contenedor j tiene capacidad suficiente para albergar a la pieza i, ésta es asignada al contenedor j y se consideraría en ítem i+1. Si, en cambio, no tuviera la capacidad se pasaría a considerar el contenedor j+1. Este proceso continuaría hasta asignar los n ítems. Esta heurística nunca reconsidera los contenedores que han sido rechazados, lo que deja espacio para la mejora.

Las dos heurísticas mencionadas en párrafos anteriores se utilizarán de la siguiente manera para el empaquetado tridimensional. En primer lugar, los ítems serán ordenados de manera decreciente por el volumen de cada uno, como la heurística BL lo hacía en 2D, pero con áreas. En segunda lugar, la heurística de Finite Next Fit será aplicada de igual forma, ya que está no varía.

## **2.2 Algoritmo 2 (NFDHDi):**

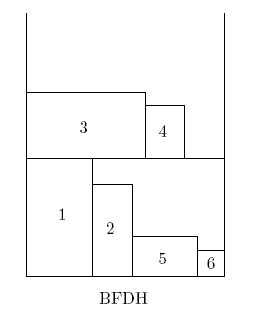
## **Heurística Next-Fit Decreasing Height (NFDH)**

Esta forma parte de la heurística basada en niveles de Strip Packing Problem, que tiene como idea central la colocación de rectángulos en niveles, es decir, cortes paralelos de guillotina en todo el ancho de la tira. Esta heurística fue propuesta por Coffman Jr en 1980. El algoritmo clasifica los ítems (rectángulos) por altura de manera decreciente, donde el ítem con mayor altura define la altura de cada nivel, y los coloca uno por uno en el nivel actual y en la posición más a la izquierda. Se cambiará de nivel cuando en el nivel actual no se tiene suficiente espacio para acomodar al siguiente rectángulo, donde este pasará al nivel superior colocado en la posición más a la izquierda. Para este caso, se realizará la implementación de esta heurística que será dirigida a problemas reales de forma tridimensional (3D), donde tanto el contenedor como los ítems serán manipulados con los ejes X, Y y Z. Esta heurística permitirá ordenar por altura los ítems y se colocará de manera descendente apegada, en primera instancia, hacia el eje Y y hacia la derecha, apilandolos hasta que ya no quepa otro ítem. Es ahí donde el siguiente ítem pasará a colocarse en la posición X que tendrá el valor del ítem con mayor largo que ha sido colocado hasta el momento y se repetirá el proceso hasta que ya no pueda entrar otro ítem en el contenedor.



## **2.3 Algoritmo 3(BFDH):**

El elemento j se empaqueta a la izquierda justificado en ese nivel, entre aquellos donde cabe, para los cuales el espacio horizontal no utilizado es mínimo. Si ningún nivel puede acomodar j, se inicializa un nuevo nivel como en NFDH**.**



# **Aporte:**

Ya implementado los algoritmos de empaquetamiento tridimensional, lo siguiente y lo más primordial es analizar la complejidad algorítmica de cada uno de los algoritmos implementados, ya que en la actualidad los algoritmos son tecnología y la tecnología es usada constantemente, por lo que un algoritmo puede estar ejecutándose muchas veces en día, dependiente el propósito con el que use, en consecuencia, existen varios factores que podrían causar un grave problema en la actualidad, tanto en la sociedad, como en el medio ambiente. En primer lugar, un algoritmo consume energía eléctrica, teniendo en cuenta esto, debemos procurar que el algoritmo que hemos desarrollado no consuma mucha energía eléctrica, ya que la energía al producirse generan residuos altamente contaminante para que el medio ambiente. En segundo lugar, un algoritmo puede ser usado para diferentes propósitos, en caso nuestro algoritmo se use por ejemplo, en hospitales, donde la salud de los pacientes depende de nuestro algoritmo, este debe funcionar correctamente al 100%, ya que si en algún momento por un error por parte del programador en algoritmo falla, este generaría un resultado erróneo que podría afectar la salud de la persona, el paciente.

# **Diseño de Aplicativo para Pruebas**

* 1. **Pseudocódigo AlgortGui:**

def Algorithm(self, rectangles, anchoC, altoC, largoC)

n <- len(rectangles)

self.calVolumen(rectangles)

self.mergeSortV(rectangles)

x <- 0

cont <- 0

base <- 0

for i in range(n)

if rectangles[i][7] > rectangles[i][5] and rectangles[i][6] >= rectangles[i][5]

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[i][2],rectangles[i][3],rectangles[i][4],rectangles[i][6],rectangles[i][5],rectangles[i][7],rectangles[i][8],cont,2)

elif rectangles[i][6] > rectangles[i][7] and rectangles[i][7] <= rectangles[i][5]

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[i][2],rectangles[i][3],rectangles[i][4],rectangles[i][5],rectangles[i][7],rectangles[i][6],rectangles[i][8],cont,2)

elif rectangles[i][6] > rectangles[i][7] and rectangles[i][7] >= rectangles[i][5]

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[i][2],rectangles[i][3],rectangles[i][4],rectangles[i][7],rectangles[i][5],rectangles[i][6],rectangles[i][8],cont,2)

elif rectangles[i][5] > rectangles[i][7] and rectangles[i][7] >= rectangles[i][6]

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[i][2],rectangles[i][3],rectangles[i][4],rectangles[i][7],rectangles[i][6],rectangles[i][5],rectangles[i][8],cont,2)

else

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[i][2],rectangles[i][3],rectangles[i][4],rectangles[i][6],rectangles[i][7],rectangles[i][5],rectangles[i][8],cont,2)

if i != 0

if rectangles[i-1][4] + rectangles[i-1][6] + rectangles[i][6] <= altoC

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[i-1][2],anchoC - rectangles[i][5],rectangles[i-1][4]+rectangles[i-1][6],rectangles[i][5],rectangles[i][7],rectangles[i][6],rectangles[i][8],cont,2)

else

if rectangles[base][2]+rectangles[base][7] + rectangles[i][7] <= largoC

rectangles[i] <- (rectangles[i][0],rectangles[i][1],rectangles[base][2]+rectangles[base][7],anchoC - rectangles[i][5],rectangles[base][4],rectangles[i][5],rectangles[i][7],rectangles[i][6],rectangles[i][8],cont,2)

base <- i

else

if rectangles[x][3] - rectangles[i][5] <= anchoC

rectangles[i] <- (rectangles[i][0],rectangles[i][1],0,rectangles[x][3] - rectangles[i][5],0,rectangles[i][5],rectangles[i][7],rectangles[i][6],rectangles[i][8],cont,2)

base <- i

x <- i

else

cont += 1

rectangles[i] <- (rectangles[i][0],rectangles[i][1],0,anchoC - rectangles[i][5],0,rectangles[i][5],rectangles[i][6],rectangles[i][7],rectangles[i][8],cont,2)

else

rectangles[i] <- (rectangles[i][0],rectangles[i][1],0,anchoC - rectangles[i][5],0,rectangles[i][5],rectangles[i][6],rectangles[i][7],rectangles[i][8],cont,2)

* 1. **Peusocpodigo NFDHDi:**

def NFDH(self, arrItem, largoBin, anchBin, altoBin)

n <- len(arrItem)

self.quicksort(arrItem, 0, n-1, 6)

bins <- 0

y <- anchBin

z <- 0

x <- 0

larC <- arrItem[0][7]

arrItem[0] <- (arrItem[0][0], arrItem[0][1], x, y - arrItem[0][5], z, arrItem[0][5], arrItem[0][6], arrItem[0][7], 0, bins, 0)

newEst <- arrItem[0][6]

for i in range(1, n)

varY <- arrItem[i-1][3] - arrItem[i][5]

if varY >= 0

arrItem[i] <- (arrItem[i][0], arrItem[i][1], x, arrItem[i-1][3] - arrItem[i][5], z, arrItem[i][5], arrItem[i][6], arrItem[i][7], 0, bins, 0)

if x + arrItem[i][7] > larC

larC <- arrItem[i][7]

else

z <- newEst

varZ <- z + arrItem[i][6]

varX <- larC + arrItem[i][7]

if varZ <= altoBin

newEst <- newEst + arrItem[i][6]

arrItem[i] <- (arrItem[i][0], arrItem[i][1], x, anchBin - arrItem[i][5], z, arrItem[i][5], arrItem[i][6], arrItem[i][7], 0, bins, 0)

if x + arrItem[i][7] > larC

larC <- arrItem[i][7]

elif varZ > altoBin and varX <= largoBin:

newEst <- arrItem[i][6]

y <- anchBin

z <- 0

x <- larC

larC <- x + arrItem[i][7]

arrItem[i] <- (arrItem[i][0], arrItem[i][1], x, y - arrItem[i][5], z, arrItem[i][5], arrItem[i][6], arrItem[i][7], 0, bins, 0)

print(arrItem[i][6])

else

newEst <- arrItem[i][6]

y <- anchBin

x <- 0

z <- 0

bins <- bins + 1

arrItem[i] <- (arrItem[i][0], arrItem[i][1], x, y - arrItem[i][5], z, arrItem[i][5], arrItem[i][6], arrItem[i][7], 0, bins, 0)

return arrItem, bins

* 1. **Peusocpodigo BFDH:**

def BFDH(self, rectangles, largoBin, anchBin, altoBin):

n=len(rectangles)

topeY=0

topeY2=0

levels=[]

highestY=-1

bins=0

topZ=0

for i in range(0,n):

rectangles[i] = (rectangles[i][0],rectangles[i][1],rectangles[i][2],rectangles[i][3],rectangles[i][4],rectangles[i][5],rectangles[i][6],rectangles[i][7],rectangles[i][8],0,2,False)

self.quicksort(rectangles, 0, n-1,7)

indexOH=-1

for i in range(0,n):

d=int(rectangles[i][6])

if d > highestY and rectangles[i][11] is False:

highestY=rectangles[i][6]

indexOH=i

rectangles[indexOH]=(rectangles[indexOH][0],rectangles[indexOH][1],largoBin-rectangles[indexOH][5],0,0,rectangles[indexOH][5],rectangles[indexOH][6],rectangles[indexOH][7],0,bins,0,True)

topeY=0

highestY=-1

topeY2=rectangles[indexOH][6]

for j in range(0,n)

levelWithSmallestResidual=None

for level in levels

if level.canFit(rectangles[j]) and rectangles[j][11] is False

if levelWithSmallestResidual is not None and levelWithSmallestResidual.availableX>level.availableX

levelWithSmallestResidual=level

elif levelWithSmallestResidual is None

levelWithSmallestResidual=level

if levelWithSmallestResidual is None and rectangles[j][11] is False

if topZ+rectangles[j][7]>altoBin:

topZ=0

for i in range(0,n):

if rectangles[i][6]>=highestY and rectangles[i][11]==False

highestY=rectangles[i][6]

indexOH=i

if topeY2+rectangles[indexOH][6]> anchBin

bins+=1

topeY=0

highestY=-1

rectangles[indexOH]=(rectangles[indexOH][0],rectangles[indexOH][1],largoBin-rectangles[indexOH][5],0,0,rectangles[indexOH][5],rectangles[indexOH][6],rectangles[indexOH][7],0,bins,0,True)

topeY=0

topeY2=rectangles[indexOH][6]

topZ=0

else:

rectangles[indexOH]=(rectangles[indexOH][0],rectangles[indexOH][1],largoBin-rectangles[indexOH][5],topeY+rectangles[indexOH][6],0,rectangles[indexOH][5],rectangles[indexOH][6],rectangles[indexOH][7],0,bins,0,True)

topeY=topeY+rectangles[indexOH][6]

topeY2=topeY2+rectangles[indexOH][6]

print("topey " + str(topeY))

highestY=-1

topZ=0

if topZ==0:

levelt=StripLevel(largoBin-rectangles[indexOH][5],topZ,topeY,bins)

else:

levelt=StripLevel(largoBin,topZ,topeY,bins)

print("topz es " + str(topZ))

print("topz es " + str(topZ))

rectangles[j]=levelt.fitRectangle(rectangles[j])

topZ+=rectangles[j][7]

levels.append(levelt)

elif rectangles[j][11] is False:

rectangles[j]=levelWithSmallestResidual.fitRectangle(rectangles[j])

Para una mejor visualización hemos decidido colocar los pseudocódigos de los algoritmos en un Folder, lo podrá encontrar fuera de este Folder.

# **Validación de Resultados y Discusión**

# **Conclusiones y Trabajos Futuros**

Las soluciones planteadas por los algoritmos de empaquetamiento en tercera dimensión, no solo tienen un impacto en la solvencia del problema, sino que va más allá con relación a los diferentes aspectos como en el contexto global, social, ambiental y económico. El impacto que conlleva los algoritmos en el ámbito global se refleja en como el software de empaquetamiento puede ser la mejor opción en la industria de cualquier enfoque de manera global que tenga relación con esta actividad, ya que aplicando el software más óptimo de empaquetamiento, se puede llegar a conseguir grande mejoras.

La solución planteada tiene diferentes “caras” en el ámbito social, ya que puede nutrir de forma positiva en la educación. La forma en q se plantea los conceptos de empaquetamiento y la utilización de los softwares más eficientes siempre pueden servir de apoyo en la solución de otros problemas que están en proceso de investigación.

La manera en cómo afecta estas soluciones en el medio ambiente tiene una perspectiva amigable en comparación con otros softwares de empaquetamiento que conllevan a un alto gasto de recursos en memoria. Se sabe que la composición de los módulos de memoria RAM contiene semiconductores de silicio y en los casos de los modelos tipo SIMM en adelante, contienen cobre en los pines; además, contienen fibra de vidrio y otros compuestos q afectan directamente con el impacto ambiental. Por ello, al generar un consumo de recursos de memoria muy bajos, aportan positivamente con el medio ambiente.

Finalmente, el impacto que tienen las soluciones propuestas en el ámbito económico se ven reflejadas en los recursos monetario q conllevan desarrollarlos, pero también, en las ventajas que se tendrían al aplicar estos softwares en empresas, industrias y otras entidades q trabajan con la problemática de la optimización del empaquetado de contenedores.

# **Conclusiones**

En conclusión, luego de analizar y comparar cada uno de los algoritmos desarrollados, se pudo observar que diferencias, ventajas y desventajas poseen cada uno de ellos respecto a otro.

1. El Algoritmo NFDHDi es más eficiente en el consumo de memoria RAM a medida que el tamaño de cajas a ordenar aumenta, respecto al algoritmo AlgortGui.
2. El Algoritmo AlgortGui es más eficiente con respecto al tiempo de ejecución con relación al algoritmo NFDHDi ya sea para 20 items como para 100 items.
3. El Algoritmo BFDH es menos eficiente pero se asegura que en la mayoría de los casos la menor cantidad de espacio sea desperdiciado

# **Anexos**

Sitio del trabajo final completo: <https://github.com/Ridaniel/WV72-CC76-ComplejidadAlgoritmica-Final>

# **Bibliografía**

[Andrea Lodi](https://www.sciencedirect.com/science/article/pii/S0166218X0100347X" \l "!), [Silvano Martello](https://www.sciencedirect.com/science/article/pii/S0166218X0100347X#!), [Daniele Vigo](https://www.sciencedirect.com/science/article/pii/S0166218X0100347X#!) (SF). Recent advances on two-dimensional bin packing problems. Recuperado de <https://www.sciencedirect.com/science/article/pii/S0166218X0100347X#SEC2>

<http://diarioecologia.com/se-crea-la-primera-memoria-ram-verde-y-ecologica/>

<http://www.scielo.org.mx/pdf/iit/v17n2/1405-7743-iit-17-02-00179.pdf>

<https://link.springer.com/article/10.1007/s10878-005-5481-6>