**UNIVERSIDAD PERUANA DE CIENCIAS APLICADAS**

**FACULTAD DE INGENIERÍA DE SOFTWARE**



**Complejidad Algorítmica**

**Integrantes:**

ASPILCUETA SOTO, Miguel Adrian

GAVILANO AURIS, Guillermo Giovanni

ARIAS DAVALOS, Rodolfo Carlos

**Profesor:** CANAVAL SÁNCHEZ, Luis Martin

**Ciclo: 2019-2**

1. **Introducción:**

En la actualidad, con los avances tecnológicos las industrias han automatizado diferentes tipos de procesos, generando así mayor eficiencia. Un ejemplo de automatización es el proceso de corte y empaquetamiento en la industria textil, la cual cuenta con diferentes tipos de algoritmos para optimizar el proceso. Es por ello, que en el presente trabajo se abordará el problema de corte y empaquetamiento, el cual consiste en realizar cortes de piezas, con las dimensiones ingresadas por el usuario, en las planchas necesarias y con el menor porcentaje de desperdicio.

Nuestra motivación para el desarrollo del proyecto de investigación es profundizar y extender nuestros conocimientos en los diferentes algoritmos que solucionan problemas de contexto real.

1. **Objetivos:**

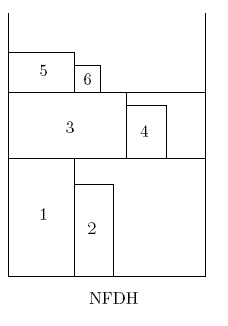
* Investigar sobre los algoritmos de corte y empaquetamiento 2D.
* Implementar un algoritmo que solucione el problema presentado.
* Analizar la complejidad algorítmica de los algoritmos investigados.

1. **Marco Teórico:**

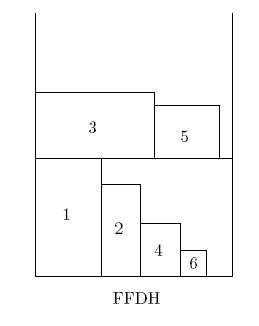
A continuación se explicará 3 algoritmos que resuelven el problema de corte y empaquetamiento.

**NFDH o Next-Fit Decreasing Height:**

El NFDH o Next-Fit Decreasing Height, es un algoritmo básico de empaquetamiento por niveles. Las n piezas rectangulares (R) se ordenan de mayor a menor altura y se coloca el primero en la parte inferior izquierda de la plancha. Luego, se evaluará cada pieza Ri donde i = {1,2,3,...n-1}, si la pieza puede caber a la derecha de la anterior, entonces se colocará, sí no, se subirá un nivel sólo si el alto de la plancha es mayor al alto del nuevo nivel. Si esta última condición no se cumple, se creará una nueva plantilla y se colocara a R en la parte inferior izquierda de la plancha.

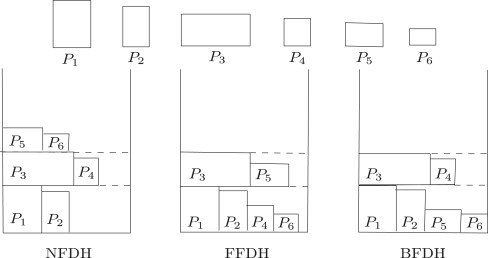


**FFDH o First-Fit Decreasing Height:**

El FFDH o First-Fit Decreasing Height, es otro algoritmo basado en el empaquetamiento por niveles. Si P = {} son todas las piezas que se desea empaquetar, lo primero a realizar es el ordenamiento descendentemente, lo segundo es acomodar la pieza a lado derecho de la posición referencial, en este caso la posición referencial es x = 0, debido a que este algoritmo coloca las piezas de izquierda a derecha, lo tercero es verificar que la pieza a colocar no pase el ancho de la plancha, es caso se pase se analiza la siguiente pieza, como su nombre lo menciona el primero en encajar será el que se colocará, si aún así no se encuentra una pieza que pueda encajar, se crea otro nivel, tomando como base de referencia el más alto del anterior nivel.

**BFDH o Best-Fit Decreasing Height:**

En este algoritmo los rectángulos se ingresan por la izquierda y luego se pegan hacia la derecha del último rectángulo empaquetado en el nivel con mínimo espacio desperdiciado. Esto significa que para colocar el siguiente rectángulo, se busca en todos los niveles existentes espacio suficiente y se calcula el área de ese espacio disponible para empacar el rectángulo que tenga área igual o cercana. Esta función se repite para todos los niveles para optimizar el espacio disponible. Tiene una similitud con el algoritmo FFDH.



1. **Análisis de la Complejidad Algorítmica:**

A continuación se detalla la complejidad algorítmica de cada una de las estrategias usadas para solucionar el problema.

Para la primera estrategia usada, NFDH, se implementó el siguiente código:

**def** NFDH(arr, an, al):

nP = 0

area = arr[0][1] \* arr[0][2]

arr[0] = (arr[0][0], arr[0][1], arr[0][2], 0, al - arr[0][2], nP)

yn = arr[0][4]

y = al

**for** i **in** range(1, len(arr)):

area += arr[i][1]\*arr[i][2]

w = arr[i-1][3] + arr[i-1][1] + arr[i][1]

**if** w <= an:

arr[i] = (arr[i][0], arr[i][1], arr[i][2], arr[i-1][3] + arr[i-1][1], y - arr[i][2], nP)

**elif** yn - arr[i][2] >= 0:

y = yn

yn = yn - arr[i][2]

arr[i] = (arr[i][0], arr[i][1], arr[i][2], 0, yn, nP)

**else** :

nP += 1

arr[i] = (arr[i][0], arr[i][1], arr[i][2], 0, al - arr[i][2], nP)

yn = arr[i][4]

y = al

**return** arr, (nP + 1), area

NFDH(MergeSort(arr), anchoP, altoP)

Se obtiene por esta función una complejidad de O(n), sin embargo el algoritmo usa un ordenamiento mergeSort de O(n log n). Por lo que el tiempo asintótico total del algoritmo es **O(n log n).**

Para la segunda estrategia usada, FFDH, se implementó el siguiente código:

def bus(arr,i,an,x):

for j in range(i+1,len(arr)):

if arr[x][3] + arr[x][1] + arr[i][1] <= an:

return j

def FFDH(arr, an, al):

nP = 0

lista=[]

arr[0] = (arr[0][0], arr[0][1], arr[0][2], 0, al - arr[0][2],nP)

yn = arr[0][4]

y = al

for i in range(1, len(arr)):

up = False

w = arr[i-1][3] + arr[i-1][1] + arr[i][1]

if w <= an:

arr[i] = (arr[i][0], arr[i][1], arr[i][2], arr[i-1][3] + arr[i-1][1], y - arr[i][2],nP)

elif w > an:

a = bus(arr,i,an,i-1)

if a != None:

aux = arr[i]

arr[i] = arr[a]

arr[a]=aux

arr[i] = (arr[i][0], arr[i][1], arr[i][2], arr[i-1][3] + arr[i-1][1], y - arr[i][2],nP)

arrAux = arr[i+1:]

arr = arr[:i+1]

arr.extend(Sort(arrAux))

continue

else:

up = True

if up:

if yn - arr[i][2] >= 0:

y = yn

yn = yn - arr[i][2]

arr[i] = (arr[i][0], arr[i][1], arr[i][2], 0, yn,nP)

else:

nP += 1

arr[i] = (arr[i][0], arr[i][1], arr[i][2], 0, al - arr[i][2], nP)

yn = arr[i][4]

y = al

return arr, (nP + 1)

FFDH(MergeSort(arr), anchoP, altoP)

Tiempo asintotico **O(n^2)**

Para la tercera estrategia usada, BFDH, se implementó el siguiente código:

def buscarBf(arr,i,an):

x = i - 1

pos = i

MIN = 0

for k in range(i + 1,len(arr)):

if(arr[x][3] + arr[x][1] + arr[k][1] > MIN and arr[x][3] + arr[x][1] + arr[k][1] <= an):

MIN = arr[x][3] + arr[x][1] + arr[k][1]

pos = k

return pos

def BFDH(arr, an, al):

nP = 0

lista=[]

arr[0] = (arr[0][0], arr[0][1], arr[0][2], 0, al - arr[0][2],nP)

yn = arr[0][4]

y = al

for i in range(1, len(arr)):

up = False

w = arr[i-1][3] + arr[i-1][1] + arr[i][1]

if w <= an:

arr[i] = (arr[i][0], arr[i][1], arr[i][2], arr[i-1][3] + arr[i-1][1], y - arr[i][2],nP)

elif w > an:

a = buscarBf(arr,i,an)

if a != i:

aux = arr[i]

arr[i] = arr[a]

arr[a]=aux

arr[i] = (arr[i][0], arr[i][1], arr[i][2], arr[i-1][3] + arr[i-1][1], y - arr[i][2],nP)

arri = arr[:i+1]

arrd = arr[i+1:]

arrd = Sort(arrd)

arri.extend(arrd)

arr = arri

continue

else:

up = True

if up:

if yn - arr[i][2] >= 0:

y = yn

yn = yn - arr[i][2]

arr[i] = (arr[i][0], arr[i][1], arr[i][2], 0, yn,nP)

else:

nP += 1

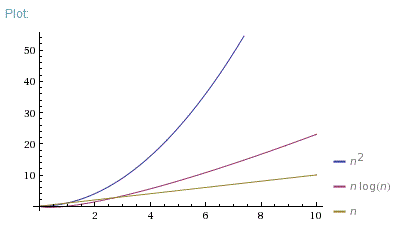
arr[i] = (arr[i][0], arr[i][1], arr[i][2], 0, al - arr[i][2], nP)

yn = arr[i][4]

y = al

return arr, (nP + 1)

Tiempo asintotico **O(n^2)**



1. **Conclusión:**

En conclusión, luego de haber investigado algoritmos de corte y empaquetamiento 2D e implementarlos, pudimos analizar la complejidad algorítmica de cada uno para poder elegir el más eficiente. El NFDH no es nada efectivo en la reducción de residuo en las planchas sin embargo tiene un tiempo asintótico de O(n log n), el FFDH mejora la reducción del residuo en las planchas. Y el BFDH mejora completamente el residuo en las planchas. Se eligió el algoritmo BFDH como la mejor opción de las investigadas.