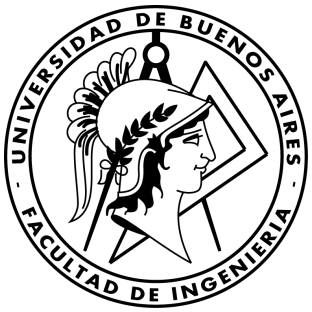


Trabajo Práctico N°2
Problema de Empaquetamiento

[75.29/95.06] Teoría de Algoritmos I
1er Cuatrimestre 2023

Emanuel, Tomas	108026
Javes, Sofia	107677

Índice

1. Demostrar que el problema de empaquetamiento es NP-Completo

El problema propuesto es ampliamente conocido en el ámbito de las Ciencias Computacionales como *BinPacking*. Dicho problema es conocido por ser $NP - Hard$ y $NP - Complete$. Primero demostraremos que el algoritmos es NP y luego haremos una reducción por parte del problema de Partición (que es conocido por ser $Np - completo$).

1.1. Verificacion NP

Para determinar que un problema es NP lo único que tenemos que hacer es verificar que una posible solución es valida en tiempo polinomial para el problema propuesto. En el caso del ejemplo propuesto simil a Bin packing, supongamos que tenemos una posible solución, siendo esta: [[0,5, 0,4], [0,8, 0,1], [0,6, 0,3]] y queremos
verificar que sea solución de la lista: [0,8, 0,5, 0,4, 0,1, 0,3, 0,6].
Determinamos:
$Target$: [0,8, 0,5, 0,4, 0,1, 0,3, 0,6]
$Source$: [[0,5, 0,4], [0,8, 0,1], [0,6, 0,3]]

Algoritmo

Una posible solución es pensar la lista $Target$ como si fuera un diccionario, vamos añadiendo todos elementos a un diccionario de clave el peso y de valor la cantidad de apariciones, teniendo en cuenta el ejemplo de arriba quedaría: $0,8 : 1, 0,5 : 1, 0,4 : 1, 0,1 : 1, 0,3 : 1, 0,6 : 1$. Ahora lo único que tenemos que hacer es verificar que los elementos de la lista $Source$ esten en el diccionario. Para continuar con esto iteramos por la lista $Source$, yendo por cada sublista, y preguntandole al diccionario:

- ¿Se encuentra en el diccionario este número?

Si la respuesta es no, devolvemos que no es una solcuión posible.

Por la contraparte, debemos restarle en 1 las iteraciones restantes al valor de la clave del diccionario. Si dicho valor al restarle se encuentra en negativo, significa que el elemento se encontraba más repetido en la lista $Source$ que en la lista $Target$.

Un ejemplo de ello podría ser [0,8, 0,5, 0,4, 0,1, 0,3, 0,6] con [[0,5, 0,4], [0,8, 0,1], [0,6, 0,3], [0,8]], quedando el diccionario: $\{0,8 : 1, 0,5 : 1, \ldots\}$.

Al hacer la búsqueda, cada valor va a quedar en cero excepto el valor de 0,8 ya que al bajar el valor nos quedaría en negativo, en ese caso, devolvemos que no es una solución posible. Cabe destacar que podría llegar el caso que la lista $Target$ sea mayor a la lista $Source$, en dicho caso deberíamos chequear si quedo algún elemento en el diccionario con valor $\neq 0$, es decir un elemento que se quedo sin usarse.

Si todas estas restricciones se cumplen, entonces es una posible solucion a nuestro problema de Bin Packing. Se adjunta un posible código:

Implementación

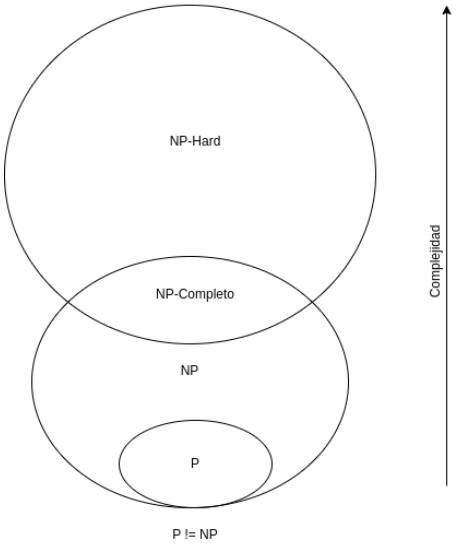
```
def esSolucion(target,current):
    dicccionario = {}
    for elemento in target:
        if elemento not in dicccionario:
            dicccionario[elemento] = 0
        dicccionario[elemento] += 1
    for sublista in current:
        for elemento in sublista:
            if elemento not in dicccionario:
                return False
            dicccionario[elemento] -= 1
            if dicccionario[elemento] < 0:
                return False
    for elemento in dicccionario:
        if dicccionario[elemento] != 0:
            return False
    return True
```

Complejidad

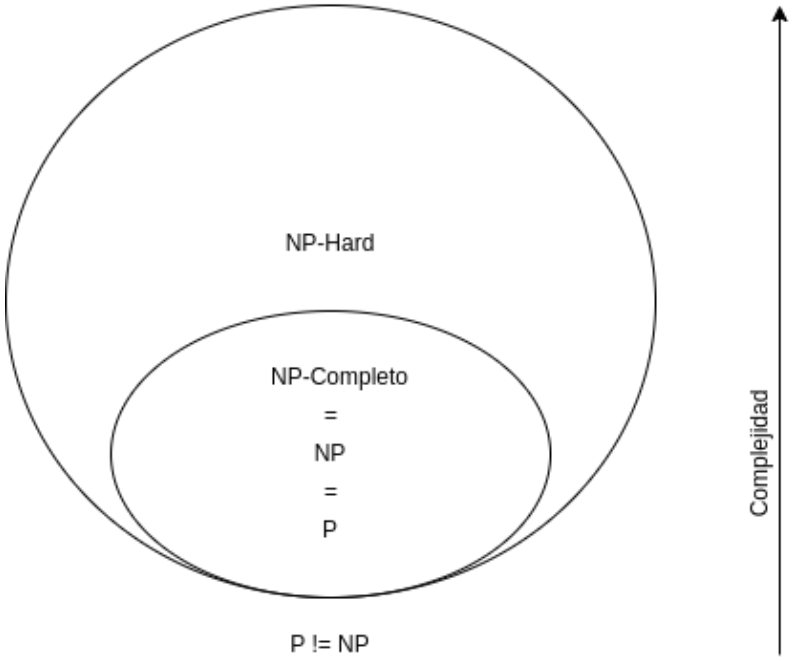
La complejidad de este algoritmo es simple, definimos n y m como la cantidad de elementos en la lista $Target$ y $Source$ respectivamente. Como hacemos un for sobre los elementos de la lista $Target$ corremos un $O(n)$, luego hacemos un for sobre la lista $Source$ (Aclaración: el total de las sublistas no importa, importa la cantidad de elementos de cada sublista, que se definió como m) siendo este $O(m)$ ya que saber si algo esta en un diccionario se considera constante. Por ultimo verificamos que ningún elemento quedo sin usarse, por lo que hacemos un $O(n)$ devuelta, terminando con una complejidad $O(n) + O(m) + O(n)$, por ende concluyendo que el algoritmo se puede verificar en tiempo polinomial, siendo este un algoritmo **NP**.

Conclusiones NP

¿Esto determinaría que el algoritmo es NP-Completo? **NO**, la familia de algoritmos NP completos esta en un nivel mas alto jerárquicamente hablando de la complejidad de problemas computacionales, un simple diagrama puede ser:



Dato: Acá se aclara que $P \neq NP$, si se llegara a confirmar que $P = NP$, se podrían resolver todos los problemas NP y NP-Completos en tiempo polinomial, el diagrama anterior se vería de este estilo:



1.2. Demostración NP-Completo

Teniendo en cuenta que Partición es un problema NP-Completo, siendo el teorema de demostración desde el otro problema NP-Completo Sub Set Sum, con demostraciones de papers: Michelle Bodnar, Andrew Lohr 2016, o University of Las Vegas paper.

Algoritmo

Podríamos pensar en una resolucion por caja negra, que resuelve problemas de Bin packing a medida que le pasamos los datos. Ahora, al tomar un problema de Particion nosotros querriamos dos subsets S_1 y S_2 que tengan la misma suma de elementos. Por un lado tenemos una implementacion por caja negra del problema de Bin packing y por el otro un arreglo que queremos verificar si se puede dividir en dos para obtener 2 arreglos separados que la suma sea igual.

Supongamos que tenemos un arreglo de n numeros, siendo cada uno de ellos $c_1, ..., c_n$ y queremos saber si los dos arreglos S_1 y S_2 donde, $\sum_{i \in S_1} c_i = \sum_{i \in S_2} c_i$. Si nosotros a cada peso que se nos da lo dividimos por la sumatoria de los pesos y lo multiplicamos por dos estariamos en el rango un rango para resolver el problema de Particion con la caja de Bin Packing, $s_i \in S$, $s_i = \frac{2 \cdot c_i}{\sum_{k=1}^{n} c_j} \in (0, 1]$ para cada i hasta n.

Ejemplo de demostracion

Supongamos que tenemos el siguiente set: $C = [2, 5, 8, 12, 11, 7, 6, 4, 10, 11]$,
con $\sum_{i=1}^{n} c_i = 76$, por lo tanto,
$S = [\frac{2 \cdot 2}{76}, \frac{2 \cdot 5}{76}, \frac{2 \cdot 8}{76}, \frac{2 \cdot 12}{76}, \frac{2 \cdot 12}{76}, \frac{2 \cdot 11}{76}, \frac{2 \cdot 7}{76}, \frac{2 \cdot 6}{76}, \frac{2 \cdot 4}{76}, \frac{2 \cdot 10}{76}, \frac{2 \cdot 11}{76}]$ teniendo
$S = [\frac{1}{19}, \frac{5}{38}, \frac{4}{19}, \frac{6}{19}, \frac{11}{38}, \frac{7}{38}, \frac{3}{19}, \frac{2}{19}, \frac{5}{19}, \frac{11}{38}]$. Dado que $\sum_{i=1}^{n} s_i = 2$, la mínima cantidad de tachos que podrían entrar los pedidos es en 2 tachos, por lo que una respuesta positiva de mínimos tachos resolvería el problema de Partición.

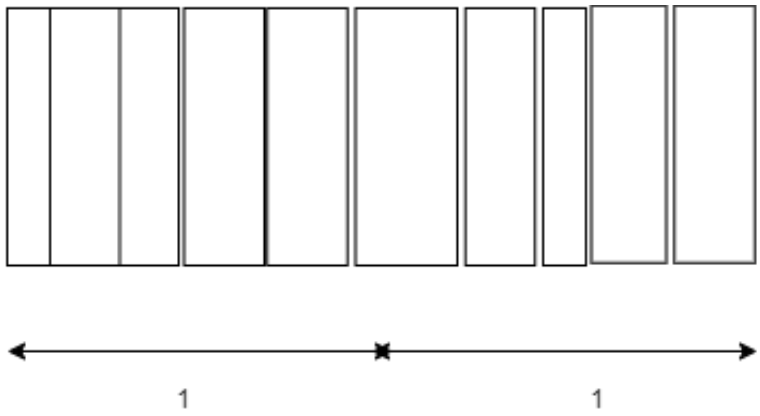


Diagrama de tachos minimos con ejemplo de arriba

Si tuviésemos una implementación por caja negra, solamente deberíamos pasarle el arreglo S y luego verificar que nos haya devuelto la mínima cantidad de tachos posible, osea 2. En el caso de que nos haya devuelto mas tachos, entonces el problema de Particion no tendria respuesta y no seria posible dividir el arreglo inicial C en dos arreglos con la misma suma de elementos. Por ultimo, para saber cuales eran los elementos originales solamente deberiamos iterar por estos dos arreglos S_1 y S_1 y hacer la conversion $\forall \frac{c_i \cdot 76}{2} \in S_1 \bigwedge \frac{c_i \cdot 76}{2} \in S_2$ y genericamente : $\forall \frac{c_i \cdot \sum_{i=1}^{n} c_i}{2} \in S_1 \bigwedge \frac{c_i \cdot \sum_{i=1}^{n} c_i}{2} \in S_2$

2. Programar el Algoritmo por Backtracking/Fuerza Bruta

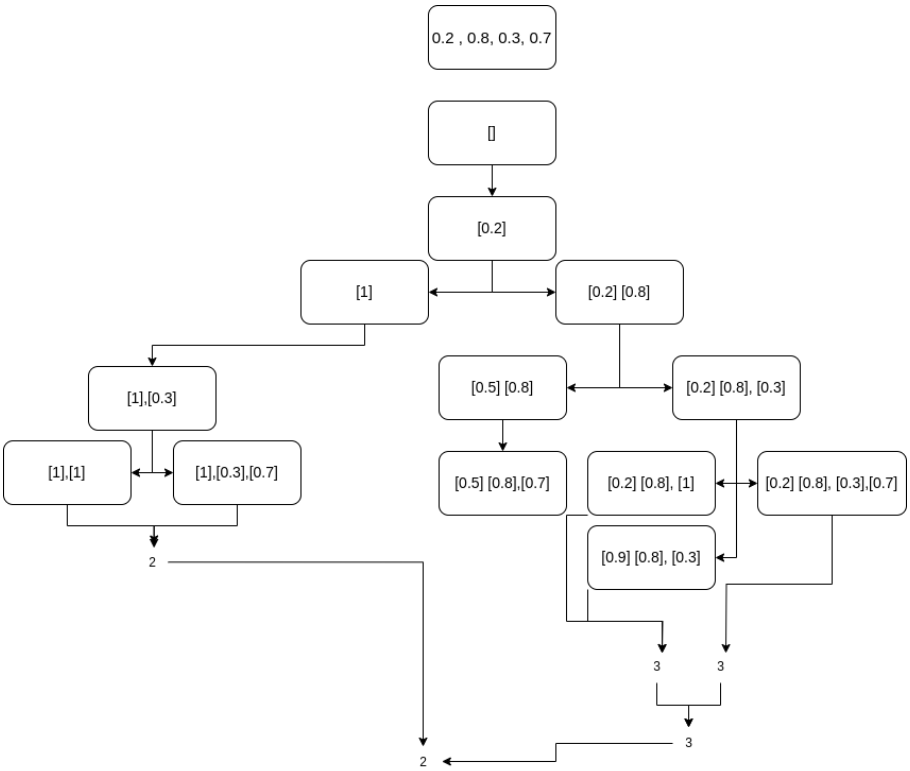
2.1. Codigo

```
def backtrackingBinPacking(weights, capacity):
    return backtrackingBinPackingAux(weights, capacity, [[weights[0]]], 1,None)


def backtrackingBinPackingAux(weights, capacity, bins, currentIndex,solucionMinima):
    if (solucionMinima and solucionMinima <= len(bins)):
        return solucionMinima
    if (currentIndex == len(weights)):
        return len(bins)
    binsForOption1 = []
    for i in range(len(bins)):
        if (weights[currentIndex] + sumBin(bins[i]) <= capacity):
            binsIOption1= bins[i] + [weights[currentIndex]]
            binsOption1 = bins[:i] + [binsIOption1] + bins[i+1:]
            option1 = backtrackingBinPackingAux(weights, capacity, binsOption1,
    currentIndex + 1,solucionMinima)
            if (solucionMinima is None or option1< solucionMinima):
                solucionMinima = option1
            binsForOption1.append(option1)
    binsOption2 = bins + [[weights[currentIndex]]]
    option2 = backtrackingBinPackingAux(weights, capacity, binsOption2 ,
    currentIndex + 1,solucionMinima)
    if (solucionMinima is None or option2< solucionMinima):
        solucionMinima = option2
    if (len(binsForOption1) == 0):
        return option2
    option1 = min(binsForOption1)
    return min(min(option1, option2),solucionMinima)

def sumBin(bin):
    sum = 0
```

```
30      for i in range(len(bin)):
31          sum += bin[i]
32      return sum
```

2.2. Seguimiento

Pensamos el algoritmo como dos posibles casos para cada item del arreglo, agregar dicho item en cada uno de los tachos existentes si este entra, y luego calcular la minima cantidad dentro de esas respuestas, o agregar el elemento en un nuevo tacho. Luego de todas las iteraciones recursivas, tenemos dos opciones posibles, la mejor opcion de agregar el item dentro de los tachos existentes y la opcion devuelta agregando el item a un tacho aparte. Un seguimiento por con un arreglo chico, ej: $[0.2, 0.8, 0.3, 0.7]$:



2.3. Poda

Una posible poda es ir guardando la solución mínima por rama. Al tener una solución mínima por rama, podemos parar de iterar cuando la cantidad de tachos que se tiene hasta el momento es mas que dicha solución mínima, una vez que la cantidad de tachos sobrepase la solución mínima, no seguimos con el algoritmo y devolvemos dicha solución mínima.

Con el codigo que tenemos ahora, tomamos como iguales los ejemplos $[[0.2, 0.8], [0.3, 0.7]]$ y $[[0.3, 0.7]], [0.2, 0.8]$ en el caso de la solucion optima. Esto es, porque vamos iterando por los indices de los arreglos y no vamos creando soluciones por cada item.

Un problema que tuvimos es el tener que copiar y popear en cada iteración por tacho ya que realizabamos un pop in-place en la lista original de los tachos. Esto nos llevo a que se aumente la complejidad.

2.4. Complejidad

Para la complejidad, como el algoritmo que propusimos itera por todas las posibles soluciones, nos quedaría una complejidad de $O(k^n)$ siendo k la cantidad de tachos posibles y t la cantidad de items. Este k estaría entre $M = \sum_{i=0}^{n} Ti$ y n (cuando $\forall t_i > \frac{1}{2}$) teniendo un tacho por cada item, quedando un $O(n^n)$ en el peor de los casos. Al hacer una poda con backtracking no estaríamos achicando la complejidad, aunque estaríamos en general achicando cuanto tiempo el algoritmo corre. Cabe decir, que todos los algoritmos NP-completos se van a poder resolver en un tiempo exponencial.

2.4.1. Memoria

En cuanto a la memoria, sabemos que todos los problemas NP-completos se encuentran en $P - Space$, es decir que utiliza una cantidad de memoria polinomial. Haciendo un desarrollo de memoria se puede llegar que este problema se encuentra en $PSpace$.

Aun teniendo n^n posibles soluciones, el algoritmo no guarda todos los posibles resultados, sino que vamos descartando a medida que pasamos por cada ramificacion. Por lo que aunque la memoria no se mantiene constante con distintos sets, se podria decir que va a haber una cantidad lineal de memoria. Por lo que, al tener esto, el espacio usado seria polinomial, ajustándose a la definición de $PSpace$.

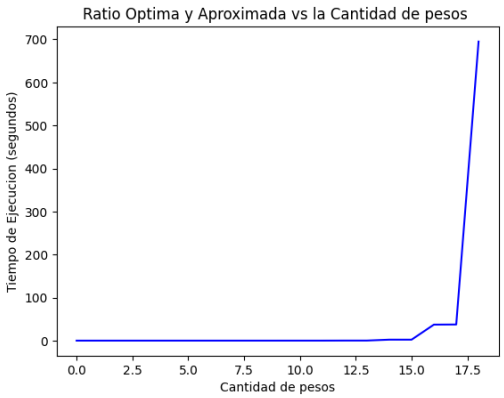
Otra posible solución para tener memoria polinomial podría ser tener una matriz de como mucho $n \cdot n$ donde vamos añadiendo cada solución de longitud a la matriz. Una solución que toma esto en cuenta es la solución por Programación Dinámica.

2.5. Mediciones

Al ser un problema NP-Hard, la complejidad aumenta exponencialmente a medida que subimos la cantidad de pesos. Para demostrar que la complejidad aumenta exponencialmente hicimos 3 tipos de tests.

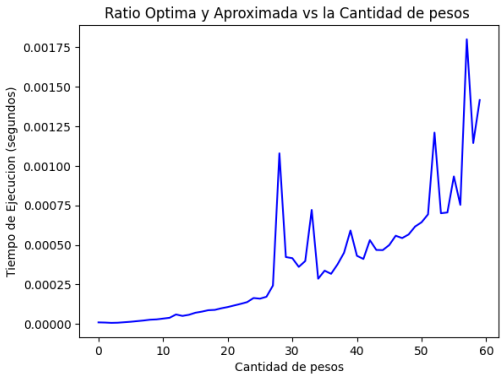
2.5.1. Test 1

Para este test quisimos probar como se comportaba el algoritmo cuando utilizabamos un arreglo de elementos para llenar los bins disponibles, es decir, ingresar n elementos de peso 0.5, por lo que vamos a tener n/2 bins. En este caso, cabe decir que la respuesta sera la misma que la de cualquier aproximación manejada en este tp, pero cambia la complejidad. A medida que aumentamos la cantidad de números, aumenta exponencialmente la complejidad del algoritmo hasta el punto que debimos bajar la cantidad de elementos porque el algoritmo no terminaba su ejecución. Aquí adjuntamos una foto de como quedo la complejidad de este test según el n:



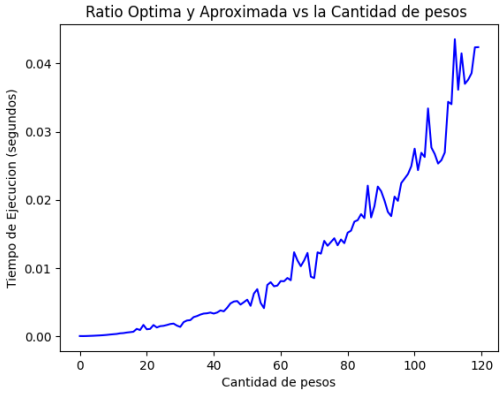
2.5.2. Test 2

Para este otro test quisimos probar como se comportaba el algoritmo con elementos que llenaran los tachos, por ejemplo elementos de peso 0.9. Ideamos este test ya que al tener un peso que sobrepasa la cantidad definida por los tachos se estaria entrando siempre en la opcion 2. Igualmente el algoritmo a medida que aumentamos la cantidad de elementos, se comporta de manera exponencial.



2.5.3. Test 3

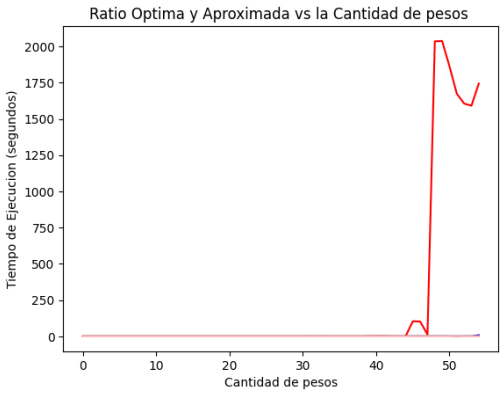
Para este test quisimos añadir en cada paso dos items que completan un tacho por si solos, como lo pueden ser 0.9 y 0.1 o genericamente $t_i + t_{i+1} = 1$.



Podemos observar como a medida que agregamos la cantidad de elementos a guardar, a pesar de que hay irregularidades y algunos saltos, a grandes rasgos, el gráfico crece exponencialmente.

2.5.4. Test 4

Ahora si, vamos probando con datasets randoms generados. Vamos a hacer esta prueba varias veces, porque puede ser el caso que justo se ordenen perfectamente los elementos y que la solucion solo tome unos segundos.

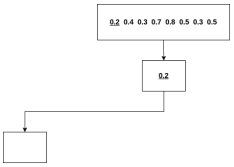


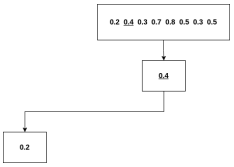
3. Aproximacion De la Catedra

3.1. Algoritmo

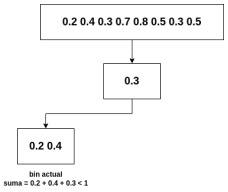
Para un primer acercamiento recomendado por la cátedra se creo un algoritmo capaz de hacer una aproximación al problema lo mas rapida posible. En ella se recorre los items una única vez y se los añade en el tacho actual. Si el item actual sobrepasa el tacho actual, entonces se desestima el tacho añadiendolo a la solucion y se abre un tacho nuevo. El algoritmo propuesto por la catedra es conocido en el campo de la computacion y resolucion de casos del problema de bin packing y su nombre es **Next Fit**

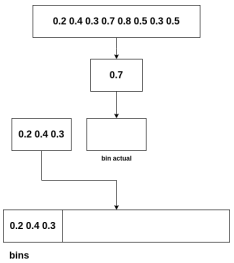
3.2. Seguimiento





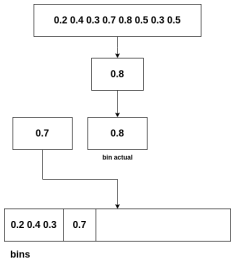
En una primera instancia se crea un arreglo de bins y un bin actual, ambos vacios. El primer elemento a analizar tiene un peso de 0.2. Al no haber ningun elemento en el bin actual se agrega este elemento de peso 0.2 ya que $0 + 0.2 \leq 1$. Continuamos con el siguiente elemento de peso 0.4. Analizamos el bin actual que ya no esta vacio por lo que debemos tener en cuenta que si agregamos el elemento nuevo no sobrepasar la capacidad maxima. En este caso como $0.2 + 0.4 \leq 1$ podemos agregarlo.

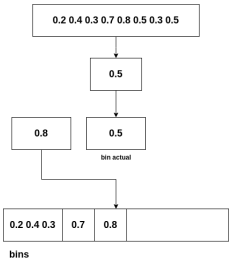




Para el analisis del tercer elemento es igual que el anterior. Como el elemento que se quiere guardar tiene un peso de 0.3 y como el tacho actual tiene capacidad disponible mayor a esta agregamos el elemento.

Para el siguiente elemento ya no podremos hacer esto mismo ya que si quisieramos agregar el elemento al tacho actual nos estariamos sobrepasando, es por eso que agregamos el bin actual a la lista de tachos y agregamos el elemento actual a un nuevo tacho (el que sera el nuevo tacho actual).

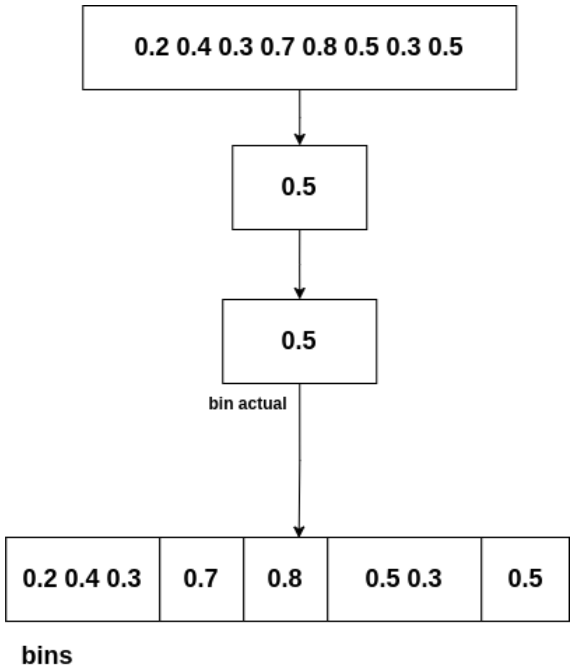




Para los siguientes elementos repetimos los pasos anteriores. En este caso el elemento a analizar tiene peso 0.8, si lo quisieramos agregar al bin actual no podriamos ya que $0.7 + 0.8 > 1$ por lo que repetimos el paso anterior. Agregamos el bin actual a la lista de bins, y asignamos como nuevo bin actual un bin que contenga el 0.8.

Para el siguiente elemento (0.5) sucedera lo mismo.

Repetimos los pasos anteriores. Agregamos el elemento si el bin actual tiene una capacidad restante mayor o igual al peso del elemento actual, sino, re-asignamos el bin actual agregando el elemento a este ultimo nuevo.



Por último, al no tener mas elementos por los que iterar, agregamos el ultimo bin actual (que contiene este último elemento) a la lista de bins.

3.3. Complejidad

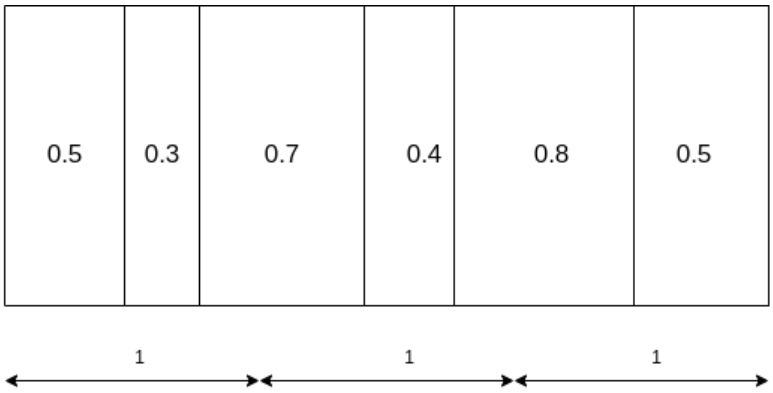
Analizar la complejidad del algoritmo Next Fit para aproximar el problema de Bin Packing

```
def next_fit(capacity,weights):
    bins = []
    bin_actual = []
    for item in weights:
        if sum(bin_actual) + item <= capacity:
            bin_actual.append(item)
        else:
            bins.append(bin_actual)
            bin_actual = [item]
    bins.append(bin_actual)
    return bins
```

Teniendo en cuenta que n son todos los pesos que se encuentran, llegamos al caso donde solamente se hace un for a través de los pesos, terminando con un simple $O(n)$. Ya que las demás operaciones son $O(1)$.

3.4. Aproximación

Para hacer esta aproximación, se podría considerar un número **mínimo** que se necesitaría para guardar los elementos. A este numero lo denotamos $M = \sum_{i=0}^{n} Ti$. M va a ser el **minimo** numero posible de tachos que se pudieran usar, ya que en el caso de usar menos estariamos dejando de lado algunos elementos, esto se podria ver con el siguiente diagrama.



Cabe destacar que en un algoritmo genérico de Next Fit, para encontrar el numero mínimo de tachos se necesitaría normalizar el numero M con la capacidad del tacho, pero en este caso al ser 1 no haría falta. Ej(capacidad C de tacho, $M = \frac{\sum_{i=0}^{n} Ti}{C}$

Ahora con la aproximacion, al tener un algoritmo que solamente itera por los pesos de los elementos, el peor caso teniendo en cuenta los pesos (m_i) de los tachos podría ser de este estilo: $m_1, m2(m_1 + m_2 > 1), m_3(m_2 + m_3 > 1)m_4 + m_3 > 1$....., un ejemplo de este caso podria ser: [0,3, 0,8, 0,3, 0,95]. Es por ello que podemos afirmar dicha hipótesis teniendo i como el numero de tachos que se usa en la aproximación:

Hipótesis: Para dos pesos contiguos, se sabrá que $m_{i-1} + m_i > 1$, por lo que para un problema completo se sabra que $m_1 + m_2 > 1$, $m_2 + m_3 > 1 m_3 + m_4 > 1$....., $m_{i-1} + m_i > 1$

Si usamos esta hipótesis, sera imposible que dos tachos contiguos esten llenos $\leq \frac{1}{2}$, solamente se abriría un nuevo si el peso del nuevo ítem se sobrepase con el peso del bin actual. Teniendo en cuenta esto y sumando las inecuaciones propuestas arriba, no tendríamos mas que $2 \cdot O$ siendo O la solución óptima de tachos a utilizar.

Otra forma de explicarlo, podria ser mirando el numero M y el numero maximo de tachos a utilizar, siendo el numero maximo directamente la cantidad de elementos n. Es por ello, que teniendo en cuenta que M siempre es $\leq$ a n, sabemos que la solucion aproximada puede ser como maximo 2 veces peor que la solucion optima.

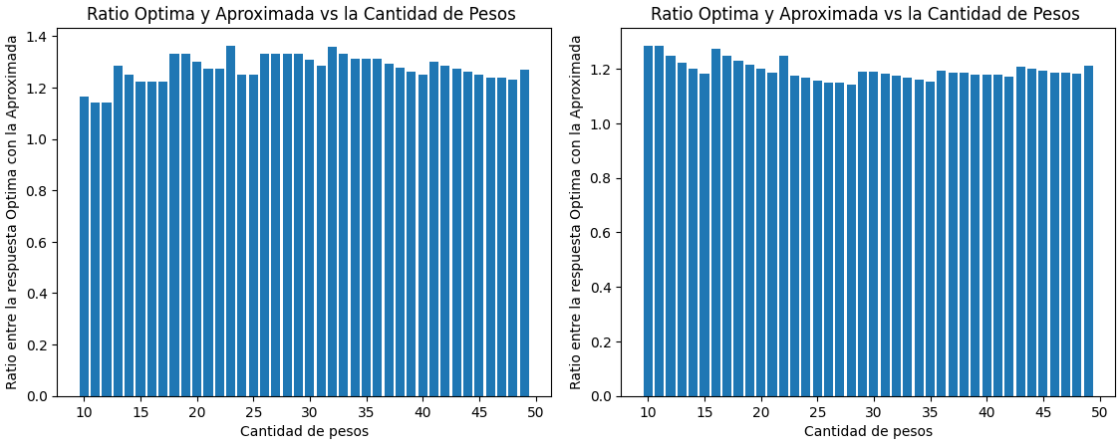
Se hallaron papers relacionados con este acercamiento en mas detalle siendo este el mas importante: Johnson, David S (1973) Massachusetts Institute of Technology, y luego este: E.G.Coffman, Jr. M.R.Garey, D.S.Johnson, Luego, se harán mejoras sobre este algoritmo utilizando otro algoritmo de aproximación llamado **First Fit**, que al igual que este tiene una aproximación $2 \cdot O$ con posibles mejoras

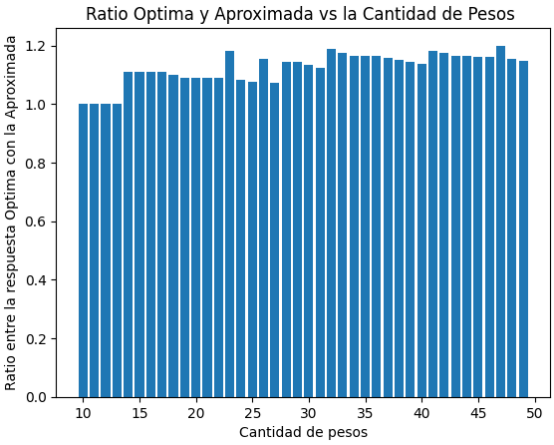
3.5. Mediciones

Para poder medir que tan bien se aproxima el algoritmo propuesto a la solucion optima ideamos 3 sets de datos con 3 tests cada uno. Para la aproximacion, para un mejor visualizacion quisimos elegir datos entre 0 y 0.5, para que no se llenaran de elementos mayores a 0.5 generando muchos tachos para pocos elementos.

3.5.1. Primer Set de Datos

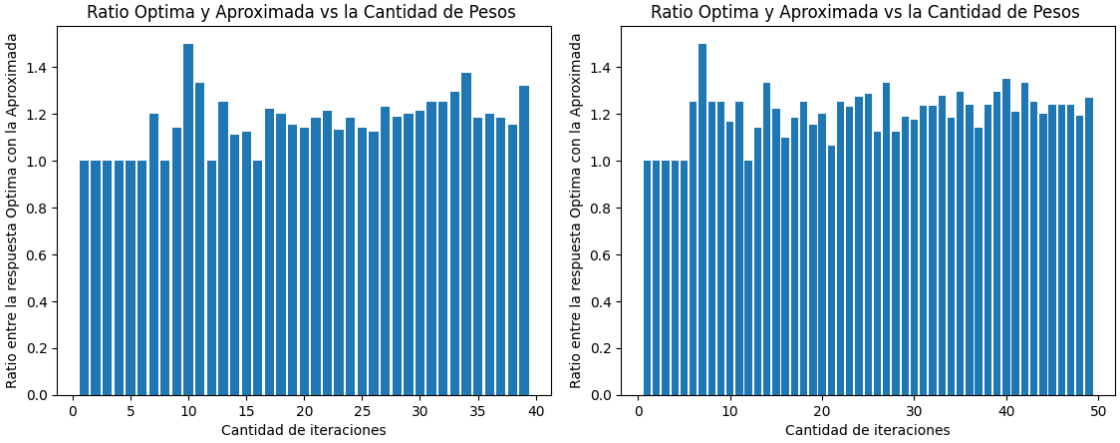
Para estos primeros 3 tests quisimos probar como se comportaba la aproximación a medida que le íbamos añadiendo números randoms a un mismo set, es decir, primero tenemos un minimo de 10 números, pudiendo ser [0.5,0.4,0.6,0.2,0.8,0.3,0.4,0.7,0.9,0.7] y de ahi le añadimos mas numeros a ese mismo set, pudiendo ser una proxima iteracion : [0.5,0.4,0.6,0.2,0.8,0.3,0.4,0.7,0.9,0.7,0.3] y asi. Quisimos probar 3 veces para ver como se comportaba en si y ver que no era coincidencia. Como podemos ver, hay casos donde el algoritmo de aproximacion se aproxima o igual o muy cerca de la solucion optima, ademas de ver que el algoritmo de aproximacion nunca es mas de 2 veces peor que la solucion optima.

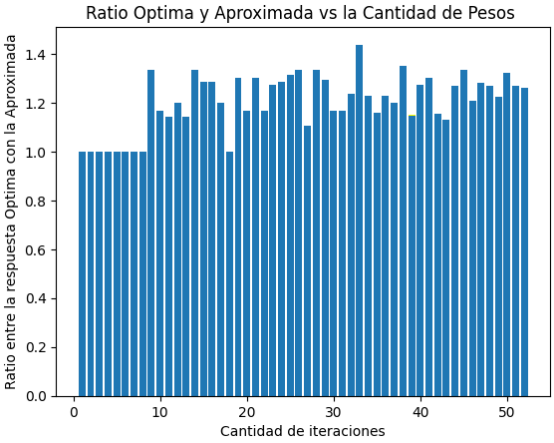




3.5.2. Segundo set de datos

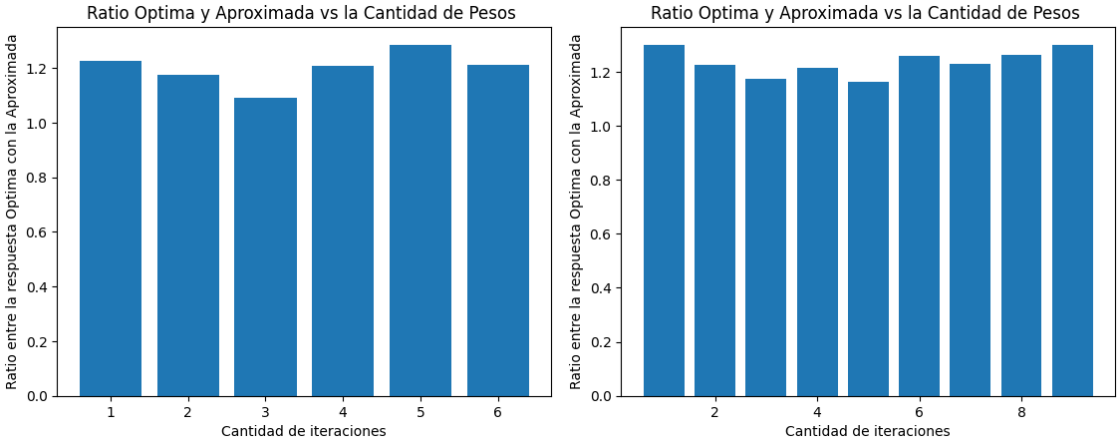
Para el segundo set de datos quisimos probar en cada paso cambiar por completo todo el set de datos. Es decir, en cada paso vamos a tener un set completamente nuevo generado por la seed random. Otra vez, pudimos ver que no se aumenta mas de $2 \cdot O$ siendo O la cantidad de tachos en la solucion Optima, aunque tampoco llega muy por ese punto, sino que se mantiene oscilando entre 1 y $\approx$1.6.

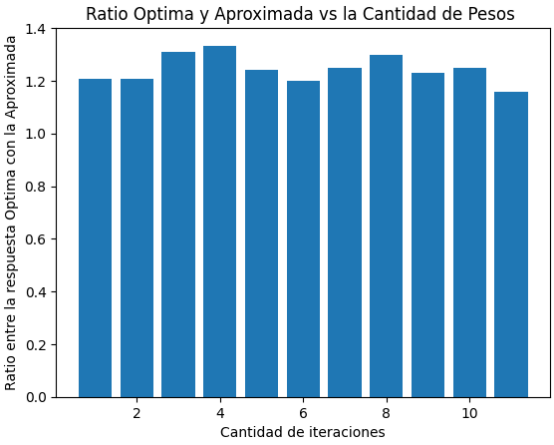




3.5.3. Tercer set de datos

Para este ultimo set de datos, quisimos mantener constante la longitud de los arreglos y probar la aproximacion un total de r(rango) veces, para poder observar como se comportaba la aproximacion.





3.5.4. Conclusión

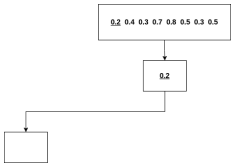
Pudimos notar, que aunque no se aproxima tan al $2 \cdot O$, a partir de las 10/11 números, la aproximación nunca es óptima. Aunque, dado estos resultados, podemos decir que esta aproximación no se encuentra tan mal por lo rápida que es.

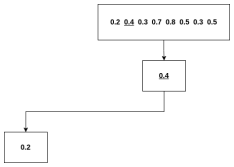
4. [Opcional] Otra aproximación de interés

4.1. Algoritmo

Para esta aproximación nos basamos en una de las primeras políticas de scheduling de los sistemas operativos. La primera aproximación es la llamada First Fit, donde, cuando se encuentre un bin con capacidad suficiente para guardar un elemento, se lo guardara ahi, y en la contra parte se lo añadiría a un nuevo bin. Un seguimiento para dicho algoritmo podría ser el siguiente:

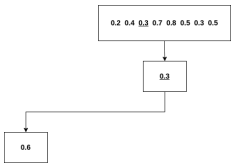
4.2. Seguimiento

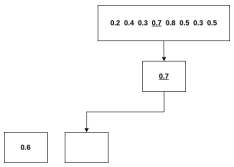




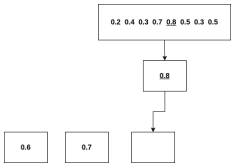
El primer elemento a analizar tiene un peso de 0.2. Al no tener cubetas, colocamos el elemento en la primer cubeta.

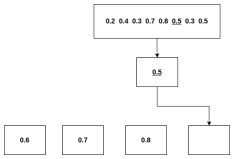
El próximo elemento tiene peso 0.4. Analizamos las cubetas que ya contienen elementos. La primera cubeta tiene un elemento de peso 0.2, ¿Podríamos poner el elemento de peso 0.4 en esta cubeta? Pues sí ya que si sumamos estos pesos estariamos dentro de la capacidad permitida por cubeta que es de 1.





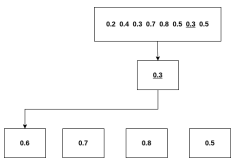
Una vez que agregamos el elemento con peso 0.4 (la primera cubeta tendrá una capacidad de uso de 0.6) analizamos el siguiente elemento. Al tener un elemento con peso 0.7 no podriamos añadirlo a la cubeta que estamos utilizando actualmente por lo que deberíamos añadir una nueva. Agregamos el elemento de peso 0.7 a esta nueva cubeta.

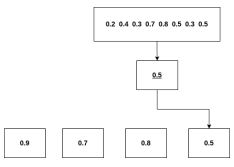




Repetimos lo mismo para el elemento de peso 0.8, al tener un peso superior a las capacidades disponibles de las 2 cubetas que tenemos, debemos agregar una nueva para poder agregar este elemento.

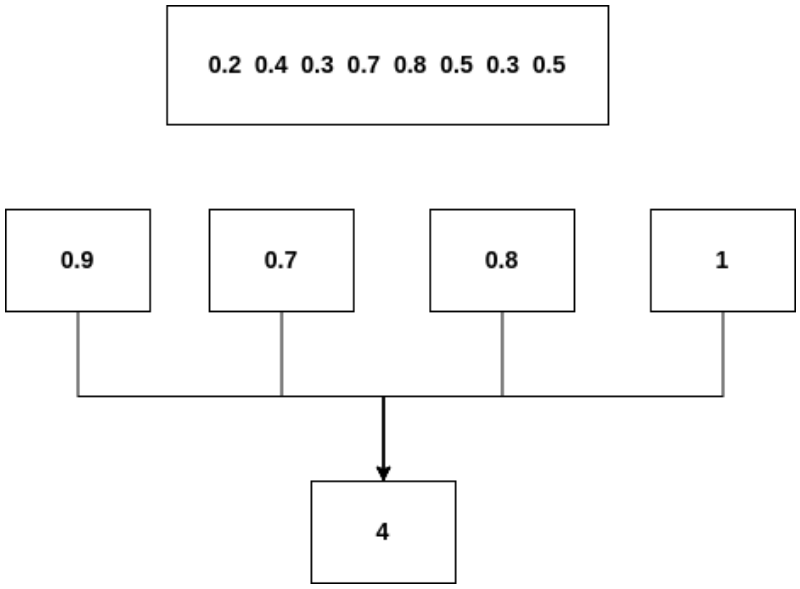
Lo mismo sucederá con el elemento de peso 0.5, como todas las cubetas que están siendo utilizadas están llenas por encima de la mitad, este elemento no podrá ser agregado a una cubeta ya existente por lo que añadimos este elemento a una cubeta nueva.





Ahora, para el elemento de peso 0.3, no hará falta añadir una nueva cubeta ya que este elemento podrá ser añadido a una cubeta que tenga una capacidad disponible de igual o mayor peso, por ejemplo, la primera cubeta que dispone de una capacidad de 0.4. Agregamos este elemento a la primera cubeta, quedando así con un peso utilizado de 0.9.

Para el último elemento realizamos lo mismo. Buscamos una cubeta que admita un elemento de peso 0.5 o más. Como disponemos de la última cubeta (con peso disponible de 0.5) añadimos el elemento.



Quedando así la aproximación del First Fit con 4 cubetas.

4.3. Complejidad

Analizando la complejidad del algoritmo del First Fit:

```
1
2  def first_fit(capacity, weights):
3      bins = []
4      for item in weights: # O(n)
5          for i in range(len(bins)): # O(m)
6              if bins[i] + item <= capacity: # O(1)
7                  bins[i] += item #O(1)
8                  break
9          else:
10             bins.append(item) # O(1)
11     return len(bins) # O(1)
```

Teniendo en cuenta que n es la cantidad de elementos de peso entre 0 y 1, y m es la cantidad de tachos que se puede guardar. Nosotros en el peor de los casos, dentro del for de tachos estaríamos buscando por todos los tachos que se pueden poner, por ejemplo, en el caso donde tengamos todos elementos con un peso mayor a 0.5, la cantidad total de tachos a utilizar si o si sera la cantidad de elementos. [0.6,0.7,0.8,0.6,0.5,0.8,0.7] la cantidad de tachos si o si sera 7. Por lo que la complejidad de este algoritmo que hace dos for seria en el peor de los casos $O(n^2)$.

4.4. Aproximación

Para empezar, nosotros planteamos que no pueden haber 2 tachos que estén hasta llenos hasta la mitad. Un ejemplo de esto, podría ser tomando los pesos $0.2, 0.5, 0.4, 0.3, 0.6, 0.5, 0.4, 0.3$ nos quedarían si o si 4 tachos teniendo peso $1.0, 1.0, 0.9, 0.3$. Directamente por como es el algoritmo, se unirían los 2 tachos llenos hasta la mitad inclusive para convertirse en uno de la sumatoria de los dos tachos. Es por ello que planteamos nuestra primera afirmación

Afirmación: La mayor cantidad de tachos llenos hasta la mitad inclusive serán hasta 1

A: respuesta de Aproximación
O : respuesta Óptima

$M = \sum_{i=0}^{n} Ti$: la suma total de pesos

Entonces, teniendo una cantidad m de tachos, sabemos que $\frac{1}{2} \cdot (A-1)$ (siendo a la respuesta de aproximación) (el menos 1 es por la afirmación) están por lo menos llenos hasta la mitad. El otro tacho también contendría algo, por lo que se afirma la ecuación: $\frac{1}{2} \cdot (A-1) \leq \sum_{i=0}^{n} Ti$ siendo Ti los pesos de cada elemento. Tomando $\sum_{i=0}^{n} Ti = M$, tenemos que $A \leq 2M + 1$. Ahora si llamamos O a la solución óptima, nosotros necesitaríamos al menos M (siendo M definida anteriormente) tachos para contener a los números. Por ejemplo, la solución óptima para el ejemplo: $0.2, 0.5, 0.4, 0.3, 0.6, 0.5, 0.4, 0.3$ la solución óptima seria $M = 3{,}8 \sim 4$ siendo estos $[[0.6, 0.4], [0.6, 0.4], [0.5, 0.5], [0.6, 0.2]]$ mientras que la solución aproximada se encuentra 5 tachos siendo estos : $[[0.2, 0.6], [0.4, 0.5], [0.6, 0.4], [0.5], [0.6]]$. Por lo que combinando las dos ecuaciones $A \leq 2 \cdot M + 1$ y $2 \cdot M + 1 \leq 2 \cdot O + 1$, por lo que $A \leq 2 \cdot O$ sin la constante del 1.

Mejora de Aproximación First Fit y Next Fit

Se puede mejorar estas dos aproximaciones, haciendo un ordenamiento de mayor a menor al principio del algoritmo. Al hacer esto, definimos los primeros tachos a los pesos mas pesados. En si, vamos a tener **por lo menos** i tachos, siendo i los elementos con peso $>$ a 0.5, esto seria porque dos elementos con peso mayor a 0.5 no estarían en el mismo tacho. Un ejemplo podría ser $0.51, 0.6, 0.7, 0.52, 0.82, 0.57$, serian si o si 6 tachos. Aunque, agregando pesos menores a 0.5 podríamos seguir teniendo mismos tachos, agregando $\Rightarrow 0.51, 0.6, 0.7, 0.52, 0.82, 0.57, 0.4, 0.3, 0.2, 0.1$ siendo en este caso $i = 6$, teniendo por lo menos 6 tachos.

Se propone un teorema determinando que First Fit Decreasing es una aproximación $\frac{3}{2}$ para el problema de Bin Packing. Universidad de Freinburg

Teniendo :

k y k^* : el numero de tachos hallados por la aproximación y la óptima respectivamente.

j : todos los tachos del $1, ..., k$,

i : item i

s_i: peso item i en el tacho j

Se coloca el item i en el tacho j que encuentre lugar, sino se aumenta el k en uno para tener un nuevo lugar. Si consideramos el tacho numero $j = [\frac{2}{3}k]$, si contiene un item i con $s_i \geq \frac{1}{2}$, entonces cada tacho anterior al j no tuvo lugar para guardar el item i. Y como tenemos un orden $s_{i'} \geq s_i \geq \frac{1}{2}$. Entonces hay por lo menos j elementos de tamaño mayor a $\frac{1}{2}$ que se tuvieron que poner en tachos individuales. Por ello

$$k^* \geq j \geq \tfrac{2}{3}k$$

Es decir, todos los j' tachos siguientes no contendrán item i con $s_i \geq \frac{1}{2}$ (por el ordenamiento mayor a menor), es por ello que los tachos $j, j+1...k$ contienen $2(k-j)+1$ items con $s_i \leq \frac{1}{2}$. En otras palabras, como k son los tachos totales, y tomamos j el numero de tacho donde no hay items con $s_i \geq \frac{1}{2}$, entonces hay **por lo menos** $2(k-j)+1$ items mas en estos tachos, pudiendo tener un tacho mas con dos items de peso 0.5 cada uno. El numero +1 considera el item dejado en el item j.

Ampliando sobre este concepto, se obtiene que la aproximación mejora en $k^* \geq \frac{3}{2}k$.

Otras mejoras

Además de esta definición, se hicieron papers obteniendo una mejor mejora de la cota de aproximación, estos son:

Johnson, David S (1973) Massachusetts Institute of Technology,

Brenda S Baker (1985),

György Dósa, Departamento de matematica Universidad de Pannonia, Hungria.

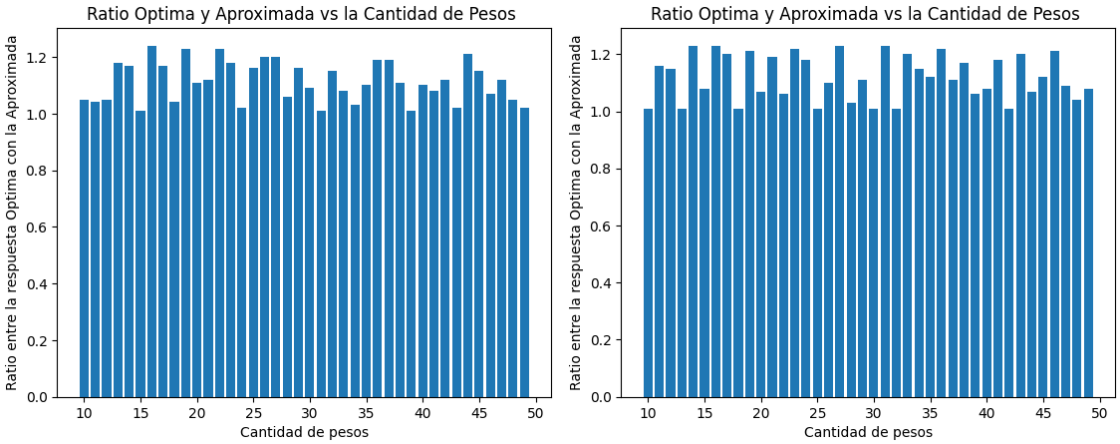
Usando una mejora de First fit, D.S Johnson demostró que se pudo llegar a una cota de $\frac{11}{9} \cdot OPT + 4$. A su vez Brenda Baker demostró que se pudo obtener una cota de $\frac{11}{9} \cdot OPT + 3$. Por ultimo el paper hecho por el departamento de matemática de la universidad de Pannonia Hungria, afirma que la cota no puede ser menor a $\frac{11}{9} \cdot OPT + \frac{6}{9}$. Para entender esta definición nos podemos

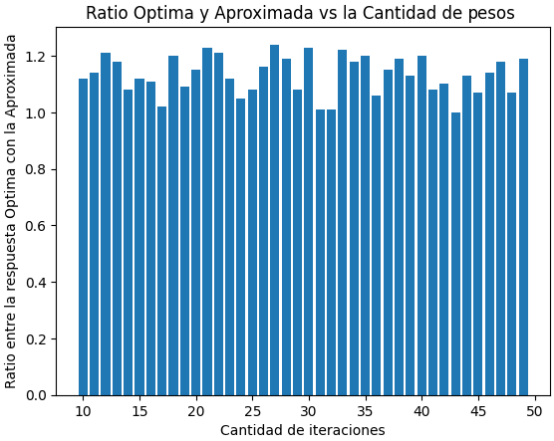
referir a un ejemplo, se dice que si la solución Óptima requiere 9 tachos, la solución por First Fit decreasing requería $\frac{1}{9} \cdot 9 + \frac{6}{9} = 12$ tachos. Aunque no extenderemos la solución mejorada de First Fit hecha por dichos papers

4.5. Mediciones Respecto del punto 3 de First Fit

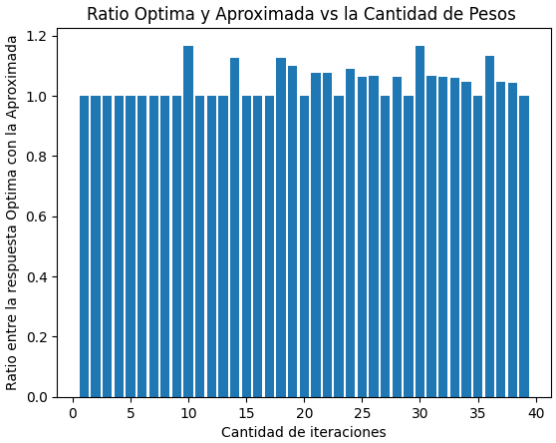
Al igual que en el punto 3, realizamos los mismos tests para First Fit acerca de la aproximacion para ver como se comportaba. Obtuvimos resultados similares a la prueba 1.

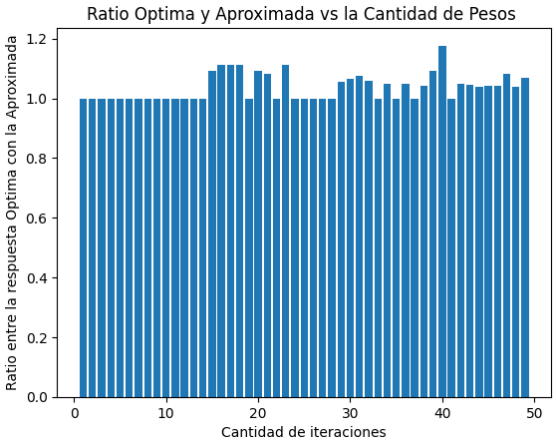
4.5.1. Primer Set de Datos

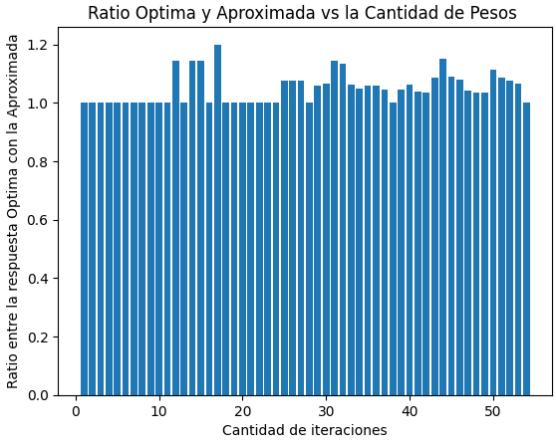




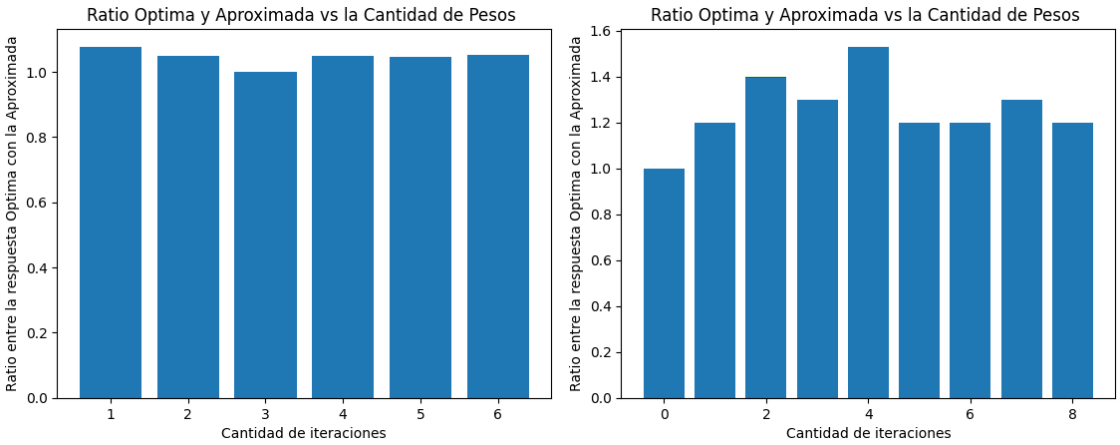
4.5.2. Segundo Set de Datos

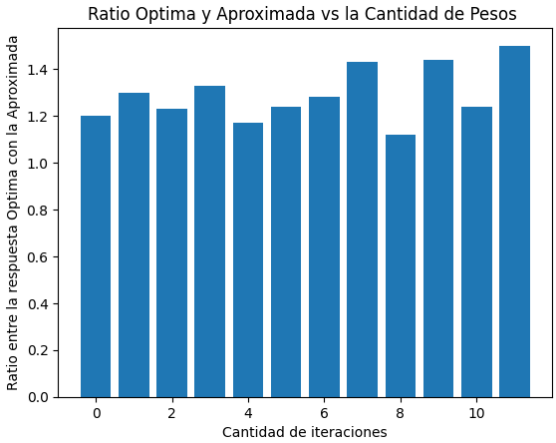






4.5.3. Tercer Set de Datos





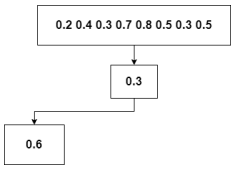
4.6. Otra Aproximación

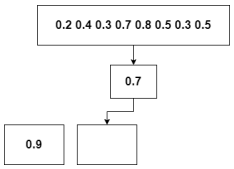
Algoritmo

Un nuevo acercamiento siguiendo las políticas de scheduling que se nos ocurrieron es un algoritmo Best Fit. La lógica de este algoritmo es la de poner un elemento de peso s_i en el tacho que mejor le vaya, es decir, en el tacho que le sobre menos lugar. Si es que no entra en ninguno, entonces lo ponemos en un nuevo tacho.

Seguimiento

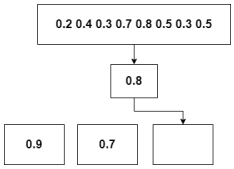
El primer elemento a analizar lo agregamos al primer bin ya que este no contiene ningún elemento. El siguiente elemento también podemos agregarlo al primer bin ya que si sumamos 0.2 + 0.4 <1 resultará en una capacidad utilizada de 0.6.

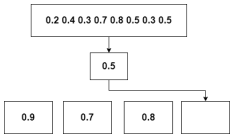




El tercer elemento al tener un peso de 0.3 también podrá ser agregado al primer bin ya que: 0.6 + 0.3 <1.

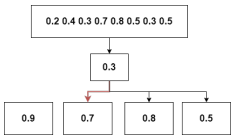
El siguiente elemento, que tiene un peso de 0.7, no podrá ser agregado ya que sobrepasaría la capacidad máxima por lo que debemos agregar un nuevo bin (vacío) y agregar el elemento a este último.

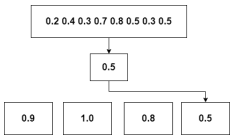




Para el siguiente elemento de peso 0.8, este deberá también ser agregado a un nuevo bin ya que las capacidades restantes de los bins ya disponibles no permiten su contención.

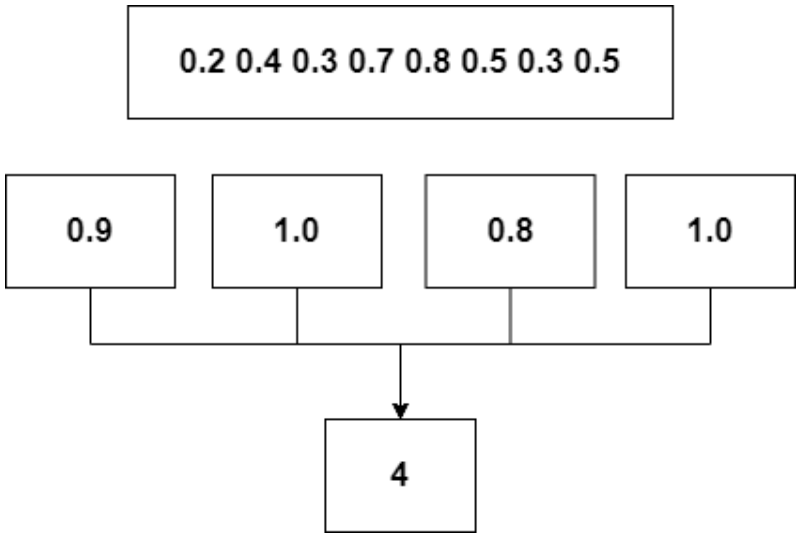
Lo mismo sucederá con el siguiente elemento de peso 0.5.





Ahora, para el elemento de peso 0.3 hay 3 opciones donde este puede ser agregado ya que las capacidades disponibles de los últimos 3 bins son iguales o mayores al peso de 0.3. Buscamos aquel que posea una menor capacidad disponibles (pero que a su vez pueda contener a este elemento). Esto sucede con el segundo bin, que tiene una capacidad disponible de 0.3. Este bin tendrá una capacidad utilizada de 1.0, la máxima posible.

El mismo razonamiento aplicará para el último elemento que tiene peso 0.5. En este caso solo tenemos 1 opción disponible, el último bin tiene una capacidad disponible de 0.5 por lo que podremos agregar el último elemento al último bin sin necesidad de agregar más bins.



El algoritmo de Best Fit resolverá el empaquetado de estos pesos con 4 bins.

Complejidad

```
1
2  def best_fit(capacity,weights):
3      bins = []
4
5      for item in weights: # O(n)
6          best_fit = capacity + 1  # ACA se pone capacidad para que en el primer caso
       que entre el primer item, se asegure entrar a la condicion de capacity - (bins
       [i] + item) < best_fit
7          best_fit_index = -1
8
9          for i in range(len(bins)):
10             if bins[i] + item <= capacity and capacity - (bins[i] + item) <
       best_fit:
11                 best_fit = capacity - (bins[i] + item)
12                 best_fit_index = i
13         if best_fit_index != -1:
14             bins[best_fit_index] -= item
15         else:
16             bins.append(best_fit_index - item)
17
18     return len(bins)
```

Al igual que en el algoritmo First Fit, nosotros iteramos por todos los pesos y luego buscamos por todos los tachos buscando el mejor. A diferencia del anterior, nosotros una vez que encontramos un tacho que pueda contener el item con peso s_i, seguimos buscando entre los siguientes tachos a ver si hay uno mejor. Aunque también, el peor caso seria que los n elementos tengan peso $\geq \frac{1}{2}$, vamos a tener tambien n tachos para guardar esos elementos, teniendo una complejidad de $O(n^2)$.

Aproximación

En 1974 ya se hizo un análisis de la que tan aproximado esta este algoritmo y se definió que el algoritmo Best Fit se encuentra en una aproximación $1{,}7O = A$ siendo O la solución óptima y A la solución aproximada. D. S. JOHNSONf, A. DEMERS, J. D. ULLMAN, (1974),