



**INSTITUTO POLITÉCNICO NACIONAL**  
**Escuela Superior de Cómputo**



**Análisis y diseño de algoritmos**

## **Ejercicio 5**

### **“Algoritmos Voraces”**

**Gil Juárez Hector David**

**3CV1**

**24 de Abril del 2024**

## Mochila fraccionaria

Este es un ejemplo llamado "El problema de la mochila fraccionaria" Dados los pesos y los beneficios(precio) de N artículos, puede ser de la forma {beneficio, peso} pon los artículos en una mochila de capacidad W de manera que obtengamos el máximo beneficio o ganancia. \*Es posible "romper/fracionar" los artículos para garantizar la ocupacion máxima de la mochila.

Entrada: arr[ ]={profit,weight}={{60,10} ,{100,20} ,{120,30} }, W=50

### Desarrollo

- Definición de la estructura Item: Se define una estructura Item que representa un artículo en la mochila. Cada artículo tiene dos atributos: profit (beneficio) y weight (peso). Además, se incluye un miembro ratio para almacenar el ratio beneficio/peso de cada artículo.
- Función de comparación compare: Se define una función de comparación para qsort que compara los ratios de dos artículos. Esta función es necesaria para ordenar los artículos en orden no creciente de ratio.
- Función fractional\_knapsack: Esta función implementa el algoritmo de la mochila fraccionaria. Recibe un arreglo de artículos, su tamaño n y la capacidad de la mochila W. Primero, calcula el ratio para cada artículo, luego ordena los artículos en orden no creciente de ratio utilizando qsort. Después, itera sobre los artículos, agregando fracciones de los mismos a la mochila hasta que la mochila esté llena o no haya más artículos. Devuelve el beneficio total obtenido.
- Función main: En la función principal, se definen los artículos de prueba y la capacidad de la mochila. Luego, se llama a fractional\_knapsack con estos valores y se imprime el resultado.

En resumen, el código define una estructura para representar los artículos, implementa una función para resolver el problema de la mochila fraccionaria utilizando un enfoque voraz y proporciona casos de prueba en la función main para demostrar su funcionamiento.


El enfoque voraz para resolver el problema de la mochila fraccionaria resulta eficaz y relativamente sencillo.

- **Eficiencia:** Este algoritmo es rápido gracias a su complejidad de tiempo  $O(n \log n)$  debido a la ordenación inicial de los artículos por su relación beneficio-peso. Es ideal para instancias con un número moderado de artículos.
- **Adecuado para fracciones:** La capacidad de tomar fracciones de los artículos lo hace adaptable a situaciones donde la división es aceptable.

- **No garantiza la solución óptima:** Aunque proporciona soluciones viables, no siempre garantiza la máxima eficiencia. En algunos casos, una estrategia voraz podría no seleccionar la combinación óptima de artículos.
- **Dependencia de la ordenación:** La eficacia del algoritmo depende de la ordenación inicial de los artículos por su relación beneficio-peso. Una ordenación inapropiada puede llevar a resultados subóptimos.
- **Flexibilidad:** Su capacidad para manejar fracciones de artículos lo hace útil en diversos contextos, como la gestión de recursos o la optimización de carteras.

En resumen, el enfoque voraz para el problema de la mochila fraccionaria ofrece una solución eficiente y adaptable, pero no garantiza siempre la máxima eficacia. En ciertos casos, otras estrategias, como la programación dinámica, podrían ser más apropiadas para garantizar soluciones óptimas.

---

 C:\Users\hecto\Downloads\ejercicio5.exe

```
El beneficio maximo obtenido es: 240.00  
  
Process returned 0 (0x0)   execution time : 0.150 s  
Press any key to continue.
```


### Caso de prueba 1

Entrada:

Artículos: {{60, 10}, {100, 20}, {120, 30}}

Capacidad de la mochila: 25

Resultado esperado: Beneficio máximo  $\approx 180$  (Fracción de los tres artículos)

 C:\Users\hecto\Downloads\ejercicio5.exe

```
El beneficio maximo obtenido es: 135.00  
  
Process returned 0 (0x0)   execution time : 0.787 s  
Press any key to continue.
```


## Caso de prueba 2

Entrada:

Artículos: {{20, 5}, {30, 10}, {10, 2}}

Capacidad de la mochila: 15

Resultado esperado: Beneficio máximo = 65 (Tomando fracciones de los artículos de ratio más alto primero)

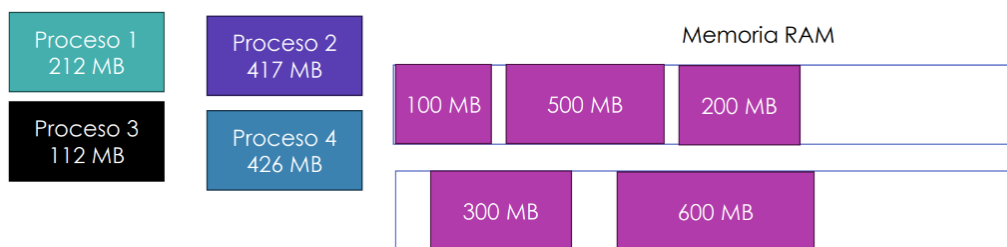
 C:\Users\hecto\Downloads\ejercicio5.exe

```
El beneficio maximo obtenido es: 54.00

Process returned 0 (0x0)   execution time : 0.821 s
Press any key to continue.
_
```

## Asignacion de bloques de memoria

- Se está diseñando un administrador de memoria para un SO. Propón una estrategia voraz, para resolver este problema.




## Restricciones

1. No se puede dividir la memoria de un proceso en dos bloques diferentes.

2. Pero, si se puede usar un solo bloque para almacenar dos o más procesos.
  - Cada que un proceso ocupe parcialmente un bloque, se deberá reducir el espacio disponible de este.

### **Solución**

1. Definición de estructuras de datos: Se definieron dos estructuras de datos principales: una para representar un proceso (Process) y otra para representar un bloque de memoria (MemoryBlock). Cada estructura contiene campos relevantes para la identificación, el tamaño y la asignación de memoria.
2. Función de asignación (allocate): Se implementó una función allocate que toma un proceso y una matriz de bloques de memoria disponibles. Esta función busca el primer bloque de memoria disponible que pueda contener el proceso sin dividirlo y lo asigna a ese bloque. Si se encuentra un bloque adecuado, se actualiza el campo allocated\_block del proceso y se reduce el espacio disponible en ese bloque.
3. Iteración sobre procesos y bloques: En la función main, se itera sobre cada proceso y se llama a la función allocate para asignarle un bloque de memoria disponible. Esto se realiza en orden, desde el primer proceso hasta el último, utilizando una estrategia voraz que asigna cada proceso al primer bloque disponible que pueda contenerlo.
4. Impresión de la asignación resultante: Después de asignar memoria a todos los procesos, se imprime la asignación resultante. Esto muestra el ID del proceso, su tamaño y el ID del bloque de memoria al que se asignó, o indica si el proceso no pudo asignarse a ningún bloque disponible.

 C:\Users\hecto\Downloads\ejercicio5-memoria.exe

```
Proceso Tamano Bloque asignado
1      212      2
2      417      5
3      112      2
4      426      No asignado

Process returned 0 (0x0)   execution time : 0.750 s
Press any key to continue.
```

### Caso adicional de prueba

#### Procesos:

Proceso 1: 300 MB

Proceso 2: 150 MB

Proceso 3: 200 MB

Proceso 4: 500 MB


#### Bloques de Memoria:

Bloque 1: 400 MB

Bloque 2: 200 MB

Bloque 3: 600 MB

Bloque 4: 100 MB

 C:\Users\hecto\Downloads\ejercicio5-memoria.exe

```
Proceso Tamano Bloque asignado
1      300      1
2      150      2
3      200      3
4      500      No asignado

Process returned 0 (0x0)   execution time : 1.060 s
Press any key to continue.
```