INSTITUTO POLITECNICO NACIONAL ESCUELA SUPERIOR DE COMPUTO

NOMBRE DEL ALUMNO:

LOPEZ MARTÍNEZ JONATHAN

MEJÍA MEJIA MAURICIO XAVIER

NAVA MONTIEL ERASMO

SILVA VÁZQUEZ DIEGO

SOLIS BUSTOS DANIEL

NOMBRE DEL PROFESOR:

GARCIA FLORIANO ANDRES

NOMBRE DE LA ASIGNATURA:

ANALISIS Y DISEÑO DE ALGORITMOS

NOMBRE DE LA PRACTICA:

ALGORITMOS VORACES

PRACTICA: 5

GRUPO: 3CV5

NOVIEMBRE 2025

INTRODUCCION

Este reporte presenta la implementación y análisis de soluciones basadas en **Algoritmos Voraces** (*Greedy Algorithms*) para resolver problemas de optimización. Esta estrategia se fundamenta en tomar la mejor decisión local en cada paso con el objetivo de alcanzar una solución global óptima de manera eficiente, evitando la complejidad de enfoques como la fuerza bruta.

Se abordarán tres problemas clásicos de la literatura computacional: el **Cambio de Monedas**, la **Mochila Fraccionaria** y la **Selección de Actividades**. Con el fin de evaluar el rendimiento y la escalabilidad de este enfoque, se desarrollaron implementaciones paralelas en **C** y **Python**, lo que permite realizar un análisis comparativo de sus tiempos de ejecución y gestión de recursos bajo diferentes volúmenes de datos.

DESARROLO

. El Problema del Cambio

Objetivo: Dar el cambio de una cantidad N utilizando la menor cantidad de monedas posibles del conjunto $\{25, 10, 5, 1\}$.

```c
int main(int argc, char const *argv[]) {
    int monedas[] = {25, 10, 5, 1}; // Definimos monedas empezando por la mayor
    int n = 4;
    int cantidad, i, num;

    // ... (Lectura de cantidad) ...

    for (i = 0; i < n; i++) {
        num = cantidad / monedas[i]; // Cantidad de monedas de ese tipo
        cantidad = cantidad % monedas[i]; // Cantidad restante por cubrir
        if (num > 0) {
            printf(" %d monedas de $%d\n ", num, monedas[i]);
        }
    }
    return 0;
}
```

Estrategia Voraz

La estrategia consiste en intentar siempre "pagar" la mayor cantidad posible con la moneda de mayor valor disponible antes de pasar a una denominación más pequeña.

Fragmentos Clave y Análisis

- Ordenamiento Implícito:

El arreglo int monedas[] = {25, 10, 5, 1}; ya está ordenado de mayor a menor. Esto es vital. Si estuviera desordenado, el algoritmo fallaría en su propósito voraz.

- **División Entera (num = cantidad / monedas[i]):**
    - **Punto Clave:** Esta línea representa la **elección voraz**. En lugar de probar combinaciones, el algoritmo toma *todas* las monedas posibles de la denominación actual ($monedas[i]$) de un solo golpe.
    - La complejidad de este algoritmo es $O(1)$ (o $O(K)$ donde K es el número de tipos de monedas), haciéndolo extremadamente eficiente comparado con Programación Dinámica.

2. Problema de la Mochila Fraccionaria

- **Objetivo:** Maximizar el valor total de los artículos en la mochila, permitiendo tomar fracciones de los objetos.

```python
# 1. Definición de la Clase Articulo
class Articulo:
    def __init__(self, id, valor, peso):
        self.id = id
        self.valor = valor
        self.peso = peso
        self.ratio = valor / peso  # Calculamos el ratio al crear el objeto

# 2. Función del Algoritmo Voraz
def mochila_fraccionaria(capacidad_maxima, articulos):
    # Ordenamos: clave=ratio, reverse=True (Descendente: Mayor a Menor)
    articulos.sort(key=lambda x: x.ratio, reverse=True)

    valor_actual = 0.0
    peso_actual = 0.0

    for item in articulos:
        if peso_actual + item.peso <= capacidad_maxima:
            peso_actual += item.peso
```

Estrategia "Divide y Vencerás" vs Voraz

A diferencia de DaC, aquí no dividimos el problema. Ordenamos los datos para tomar la decisión óptima inmediata.

1. El Criterio de Selección (Greedy Choice Property):

```python
self.ratio = valor / peso
articulos.sort(key=lambda x: x.ratio, reverse=True)
```

 Punto Clave: El "truco" de este problema es la densidad de valor. No tomamos el objeto más valioso (podría ser muy pesado) ni el más ligero (podría valer poco). Tomamos el que tiene **más valor por unidad de peso**.

 En C (mochilaP5.txt), esto se logró con qsort y una función comparar. En Python, se usa sort con una función lambda, aprovechando el algoritmo **Timsort** ($O(N \log N)$).

```
proporcion = espacio_restante / item.peso
valor_actual += item.valor * proporcion
```

Punto Clave: Esta línea diferencia la mochila fraccionaria (Voraz, fácil) de la mochila 0/1 (NP-Completo). Al poder dividir, siempre podemos llenar la mochila al 100% de su capacidad con el mejor valor disponible.

3. Selección de Actividades

Objetivo: Seleccionar el máximo número de actividades compatibles (que no se traslapen en tiempo).

```c
typedef struct { int inicio; int fin; } Actividad;

// Función de ordenamiento (Bubble Sort simplificado)
void ordenar_actividades(Actividad actividades[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (actividades[j].fin > actividades[j+1].fin) {
                // Swap
                Actividad temp = actividades[j];
                actividades[j] = actividades[j+1];
                actividades[j+1] = temp;
            }
        }
    }
}

Actividad* seleccion_actividades(Actividad actividades[], int n, int* cantidad) {
    // ...
    ordenar_actividades(actividades, n);
```

Estrategia Voraz

La clave es priorizar las actividades que **terminan antes**.

Fragmentos Clave y Análisis

1. Ordenar por tiempo de finalización (fin):

- **Punto Clave:** Si ordenamos por inicio o por duración, el algoritmo falla. Al ordenar por tiempo de finalización (actividades[j].fin), liberamos el recurso (la

sala, el procesador) lo antes posible, maximizando el tiempo restante para otras actividades.

2. El Bucle de Selección:

```
if (actividades[i].inicio >= ultimo_fin) {
    // Seleccionar y actualizar ultimo_fin
}
```

Punto Clave: Esta comparación lineal ($O(N)$) es suficiente una vez que los datos están ordenados. No se requiere *backtracking* (volver atrás), lo que hace al algoritmo muy rápido.

Reporte de Resultados y Métricas

Se realizaron pruebas de ejecución para medir el rendimiento de los algoritmos con diferentes tamaños de entrada (N).

N (Artículos)	Tiempo Python (s)	Tiempo C (s)	Análisis
10	0.000015	0.000002	Ambos instantáneos.
100	0.000085	0.000012	C mantiene una ventaja de ~7x.
1000	0.000850	0.000150	La complejidad $O(N \log N)$ es evidente y escalable.

Prueba de Actividades (Resultados de Consola):

Plaintext

```
=== ALGORITMO VORAZ - SELECCION DE ACTIVIDADES ===

Caso 1: Actividades variadas (N=11)
Seleccionadas: 4 (36.4%)
Tiempo: 0.000003 segundos

Caso 2: Sin traslape (N=5)
Seleccionadas: 5 (100.0%) -> Eficiencia máxima

Caso 3: Completamente traslapadas (N=5)
Seleccionadas: 1 (20.0%) -> El algoritmo elige correctamente solo una.
```

Conclusiones

1. **Eficiencia:** Los algoritmos voraces mostraron tiempos de ejecución en el orden de microsegundos incluso para . La complejidad dominante es la del ordenamiento ().
2. **Comparativa de Lenguajes:** La implementación en **C** gestiona la memoria manualmente (malloc/free) y resulta más rápida en tiempo de ejecución bruto. **Python**, mediante su método sort() (Timsort) y gestión automática de memoria, ofrece un código más limpio y legible con una penalización de rendimiento aceptable para estas magnitudes.
3. **Correctitud:** En los tres problemas probados (Cambio, Mochila, Actividades), el enfoque voraz garantiza la solución óptima global debido a la naturaleza de los problemas (subestructura óptima y propiedad de selección voraz).