.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

# Trabajo Práctico Parte 1

8 de febrero de 2025


| Nombre | Padrón |
|---|---|
| Maximo Giovanettoni | 110303 |
| Franco Bossi | 111122 |
| Luca Saluzzi | 108088 |
| Agustin Conti | 107800 |
| Gabriel Peralta | 101767 |

# 1. Algoritmo

El problema presentado es el siguiente: Se tiene una fila de n monedas con diferentes valores, en cada turno uno de los dos jugadores elije entre agarrar la primera moneda de la fila o la ultima. El que acumula mas valor es el ganador. En este caso en particular, si bien hay dos jugadores, solo Sophia elige monedas. Es decir, que en su turno toma una moneda para él y en el próximo una moneda para el jugador rival.

Se nos pedia diseñar un algoritmo greedy que haga que el jugador 1 (Sophia) siempre gane, terminando siempre con un valor total acumulado mayor al del jugador 2 (Mateo).

El algoritmo planteado es el siguiente:

## 1.1. Jugador 1 elige moneda

El jugador1, en su turno, debe elegir la moneda mas grande, por lo que compara la del principio y la del final y agarra aquella de mayor valor. En nuestra implementación, en vez de "sacar.esa moneda de la pila (que vendría a ser equivalente a hacer pop de la lista), simplemente avanza el indice correspondiente, que puede variar según si eligió la primera o la ultima moneda.

```
if (jugador == JUGADOR1 and monedas[izq] > monedas[der]):
    return monedas[izq], izq + 1, der
else:
        return monedas[der], izq, der - 1
```

## 1.2. Jugador 1 elige la moneda del Jugador 2

En el turno del Jugador 2, es el Jugador 1 el que vuelve a elegir la moneda, pero en este caso para su rival. Es por esto que deberá nuevamente comparar la moneda inicial con la final y agarrar la de menor valor. Al igual que en el caso anterior, se avanza el indice correspondiente.

```
    if (jugador == JUGADOR2 and monedas[izq] < monedas[der]):
        return monedas[izq], izq + 1, der
    else:
        return monedas[der], izq, der - 1
```

## 1.3. Loop

Estos dos pasos anteriores se repetirán hasta que la posición inicial sea mayor a la final, es decir, hasta que los indices final e inicial estén en la misma posición.

## 1.4. Algoritmo completo y complejidad algorítmica

En este caso particular, Sophia es el jugador 1 y Mateo el jugador 2

```
SOPHIA = 0
MATEO = 1

def elegir_moneda(monedas, jugador, izq, der):
    if (jugador == SOPHIA and monedas[izq] > monedas[der]) or (jugador == MATEO and
     monedas[izq] < monedas[der]):
        return monedas[izq], izq + 1, der
    else:
        return monedas[der], izq, der - 1

def greedy_monedas(monedas):
    izq, der = 0, len(monedas) - 1
    puntos_sophia, puntos_mateo = 0, 0

    turno_sophia = True
```

```
    while izq <= der:
        jugador = SOPHIA if turno_sophia else MATEO
        moneda, izq, der = elegir_moneda(monedas, jugador, izq, der)
        if turno_sophia:
            puntos_sophia += moneda
        else:
            puntos_mateo += moneda
        turno_sophia = not turno_sophia

    return puntos_sophia, puntos_mateo
```

Tal y como se aprecia en el código, este algoritmo trabaja en O(n) ya que se recorre una sola vez cada posición de la lista de monedas, avanzando el indice cuando es necesario. Esto se podrá comprobar más adelante en conjunto con los gráficos correspondientes.

### 1.5. Regla greedy del Algoritmo

Un algoritmo se considera greedy si construye una solución paso a paso mediante la elección localmente óptima en cada iteración, sin considerar consecuencias futuras ni reevaluar decisiones previas. En este problema:

Elección Greedy de Sophia (Jugador 1): En su turno, Sophia siempre elige la moneda de mayor valor disponible (primera o última de la fila). Esto maximiza su ganancia inmediata y refleja una decisión localmente óptima.

Elección Greedy para Mateo (Jugador 2): En el turno de Mateo, Sophia (actuando por Mateo) elige la moneda de menor valor disponible, minimizando la ganancia de Mateo. Esta elección también es localmente óptima para perjudicar al rival.

Su optimalidad depende de que las decisiones locales aseguren una ventaja global acumulada, lo cual se demuestra formalmente a continuación.

## 2. Optimalidad del Algoritmo

### 2.1. Demostración por Inducción Fuerte

Sea $S(n)$ la afirmación:

"Para cualquier fila de $n$ monedas, el algoritmo greedy asegura que puntos_Sophia $>$ puntos_Mateo."

#### Base Inductiva

**Caso $n = 1$:**

Sophia toma la única moneda:

$$\text{puntos\_Sophia} = v_1, \quad \text{puntos\_Mateo} = 0.$$

Claramente, se cumple $v_1 > 0$.

**Caso $n = 2$:**

Sophia elige la moneda de mayor valor, y Mateo recibe la restante:

$$\text{puntos\_Sophia} = \text{máx}(v_1, v_2), \quad \text{puntos\_Mateo} = \text{mín}(v_1, v_2).$$

Es evidente que:

$$\text{máx}(v_1, v_2) > \text{mín}(v_1, v_2).$$

## Hipótesis Inductiva

Supongamos que $S(k)$ es verdadera para todo $k < n$, es decir, el algoritmo funciona para cualquier fila con menos de $n$ monedas.

## Paso Inductivo

Considérese una fila de $n$ monedas:

$$[v_1, v_2, \ldots, v_n].$$

**Turno de Sophia:**

Elige máx$(v_1, v_n)$. Supongamos sin pérdida de generalidad que $v_1 > v_n$, entonces Sophia toma $v_1$, dejando la fila restante:

$$[v_2, \ldots, v_n].$$

**Turno de Mateo:**

Sophia elige para Mateo mín$(v_2, v_n)$. Sea:

$$v_{\text{mateo}} = \text{mín}(v_2, v_n).$$

La fila restante tiene $n - 2$ monedas.

Por hipótesis inductiva, el algoritmo garantiza que Sophia superará a Mateo en la sub-fila de $n - 2$ monedas. La ventaja acumulada después de estos dos turnos es:

$$\text{Diferencia} = (v_1 - v_{\text{mateo}}) + \text{Diferencia}(n - 2).$$

Dado que $v_1 > v_{\text{mateo}}$ (por elección greedy) y Diferencia$(n - 2) \geq 0$ (por hipótesis inductiva), se cumple que:

$$\text{Diferencia} > 0.$$

## Conclusión

Por inducción, $S(n)$ es verdadera para todo $n \geq 1$. Por lo tanto, el algoritmo greedy asegura que Sophia siempre gane.

# 3. Análisis de variabilidad y mediciones

Para comparar como afectan los distintos valores de las monedas (no la cantidad, si no los valores en si), medimos varias ejecuciones con sets de datos de alta y baja variabilidad.

```
VARIABILIDAD_BAJA = 10
VARIABILIDAD_ALTA = 1000000

# Tama os del array
x = np.linspace(1000, 10_000_000, 25).astype(int)

# Generador de monedas aleatorias
def get_random_cant_monedas(s, var):
    return [random.randint(1, var) for _ in range(s)]

# Medimos los tiempos de ejecuci n
results_low_var = time_algorithm(greedy_monedas, x, lambda s: [
    get_random_cant_monedas(s,VARIABILIDAD_BAJA)])
results_high_var = time_algorithm(greedy_monedas, x, lambda s: [
    get_random_cant_monedas(s,VARIABILIDAD_ALTA)])
```

Tal y como se puede observar en los siguientes gráficos. El algoritmo se comporta en forma O(n), con un ajuste muy bueno con respecto a las rectas en ambos casos y con un error pico del 0.2 en los casos de mayor variabilidad. Podríamos concluir entonces que, para los casos de mayor variabilidad, el error se maximiza y se diferencia para tamaños de arrays mayores.

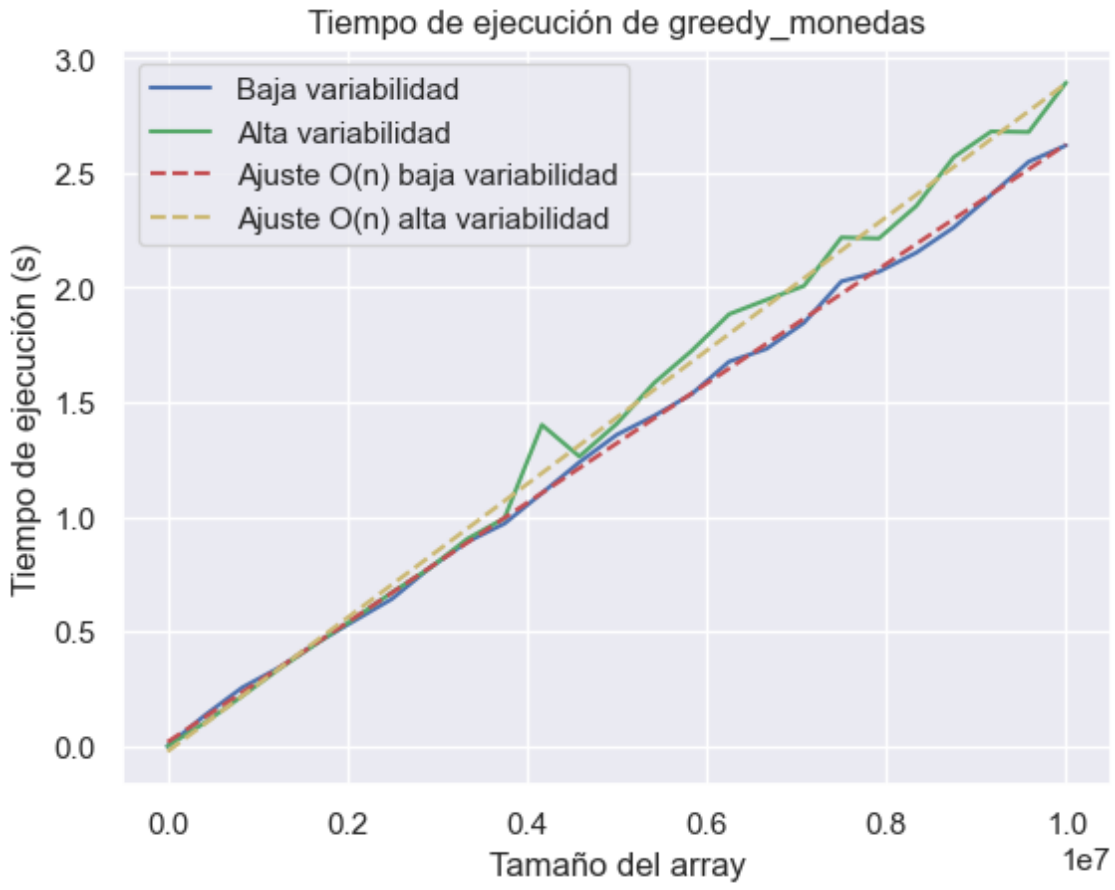


Figura 1: Tiempo de Ejecución del algoritmo con alta y baja variabilidad comparado con rectas de ajuste

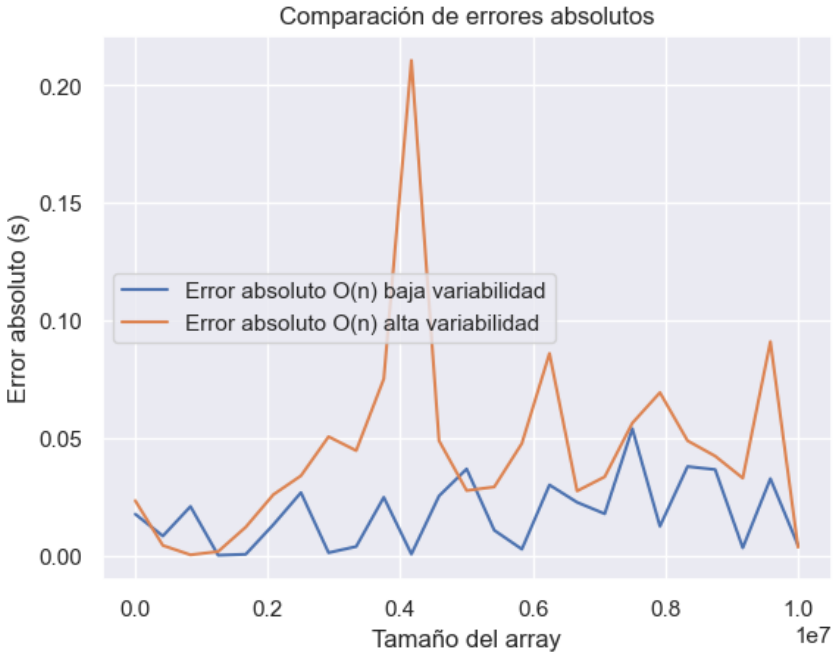


Figura 2: Errores absolutos del algoritmo según tamaño del array con alta y baja variabilidad

Notamos que suele haber mas error con variabilidad alta, pero no siempre es el caso. Lo probamos 20 veces con los parámetros mostrados y resultó en que el 60 % de las veces variabilidad alta tenia más error.