Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico: Juego de Hermanos
Informe

Segundo Cuatrimestre 2024

Lucas Ricardo Williams
103018

Gaston Proz
105868

Índice

1. Primera parte: Introducción y primeros años

1.1. Análisis del problema

Dado el problema del juego de las monedas, donde Sophia siempre elige la mayor moneda en su turno y la menor moneda en el siguiente para su hermano, Mateo, podemos decir que el algoritmo de la solución a este problema, sigue la misma lógica y que por siempre tomar el mayor y por siguiente menor en cada instancia, se obtiene una solución local con la esperanza de que sea una solución optima al problema general. El algoritmo en esta primera parte esta clasificado como Greedy. Se analizara posteriormente la optimalidad de la solución.

1.2. Análisis de optimalidad del algoritmo Greedy

El algoritmo provisto es optimo. A continuación vamos a analizar la razón, desestimando el caso en el que haya una cantidad par de monedas del mismo valor, cuyo caso el resultado del juego seria empate sin importar la estrategia a utilizar.

La justificación de la optimalidad del algoritmo reside en que la solución local sea optima en cada instancia. Para ello, la analizaremos.

En primera instancia, Sophia obtiene la moneda $M_{turno,s}$ con mayor valor entre la primera y la ultima. Por consiguiente se elige la moneda $M_{turno,m}$ para Mateo que es la mínima entre la primera y la ultima. Para cada moneda elegida hay diferentes escenarios.

- En el turno de Sophia, se obtiene el mayor entre las dos opciones, por lo que la moneda de Sophia siempre sera mayor a la de Mateo.
- En el turno de Mateo, se obtiene la menor moneda, cual esta asegurado que es menor que la ultima moneda obtenida por Sophia ya que su turno fue antes. Se agrega una moneda mas que puede ser mayor a la de Sophia en cuyo caso no se obtiene, u otra menor y en ese caso seria la resultante. Esta asegurado que la moneda de Mateo siempre es menor a la de Sophia.

En otras palabras tenemos que $M_{i,s} > M_{i,m}$ y $M_{i,m} < M_{i+1,s}$. Por otra parte, la sumatoria de Sophia siempre seria mayor que la de Mateo $S_{sophia} > S_{mateo}$ y la sumatoria de Sophia maximiza el valor que se puede obtener con las reglas del juego, así mismo la sumatoria de Mateo minimiza el valor que se puede obtener.

1.3. Algoritmo Greedy

A continuación se muestra el código de solución al problema.

```
def juego_monedas(monedas):
    solucion = []
    suma_sophia = 0
    suma_mateo = 0
    turno_sophia = True

    monedas = deque(monedas)

    append_solucion = solucion.append  # Local variable for faster access

    append_solucion = solucion.append  # Local variable for faster access
    while monedas:
        if turno_sophia:
            valor_moneda, resultado = obtener_mayor_moneda(monedas)
            suma_sophia += valor_moneda
            append_solucion(resultado)
        else:
            valor_moneda, resultado = obtener_menor_moneda(monedas)
            suma_mateo += valor_moneda
            append_solucion(resultado)
```

```
        turno_sophia = not turno_sophia

    return solucion, suma_sophia, suma_mateo
```

La complejidad del algoritmo propuesto es $\mathcal{O}(n)$, debido que se recorre el arreglo de monedas una única vez y en cada turno se llama a obtener mayor moneda o menor moneda lo cual tiene una complejidad de $\mathcal{O}(1)$ dado que solo se saca la primera moneda o la ultima. Esto se corrobora en la sección de mediciones con la figura 1 y 2 correspondientes.

1.4. Análisis de optimalidad

En instancias de alta variabilidad y baja variabilidad del valor de las monedas de, la solución del algoritmo Greedy sigue siendo optima, el resultado final maximiza el resultado para Sophia y minimiza para Mateo, siguiendo las reglas del juego, de este modo Sophia siempre gana y Mateo siempre pierde.

1.5. Mediciones Primera Parte

Se realizaron mediciones en base a crear arreglos de diferentes largos, con una cantidad mínima de 100 y máxima de 50.000, donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`). Se realizo el calculo del error cuadrático medio entre los errores de los ajustes para comprobar numéricamente cual ajuste se asemeja mas a la complejidad real del algoritmo.

El siguiente código corresponde al gráfico de figura 1, donde se muestra el ajuste cuadrático, lineal y la medición real que se tomo.

```
def get_random_array_s(size: int):
    return np.random.randint(1, 100000, size).tolist()
# Siempre seteamos la seed de aleatoridad para que los # resultados sean
    reproducibles
seed(12345)
np.random.seed(12345)
sns.set_theme()

# La variable x van a ser los valores del eje x de los graficos en todo el notebook
# Tamano m nimo=100, tama o maximo=100.000 , cantidad de puntos=20
x = np.linspace(100, 100_000, 50).astype(int)

results_greedy = time_algorithm(juego_monedas, x, lambda s: [get_random_array_s(s)
    ])

f_n = lambda x, c1, c2: c1 * x + c2
f_n2 = lambda x, c1, c2: c1 * x**2 + c2

c_iter, _ = sp.optimize.curve_fit(f_n, x, [results_greedy[n] for n in x])
c_n2, _ = sp.optimize.curve_fit(f_n2, x, [results_greedy[n] for n in x])

ax: plt.Axes
fig, ax = plt.subplots()
ax.plot(x, [results_greedy[n] for n in x], label="Medicion")
ax.plot(x, [f_n(n, c_iter[0], c_iter[1]) for n in x], 'g--', label="Ajuste $n$")
ax.plot(x, [f_n2(n, c_n2[0], c_n2[1]) for n in x], 'y--', label="Ajuste $n^2$")
ax.set_title('Tiempo de ejecucion de algoritmo greedy')
ax.set_xlabel('Tamano del array')
ax.set_ylabel('Tiempo de ejecucion (s)')
ax.legend()
None
```

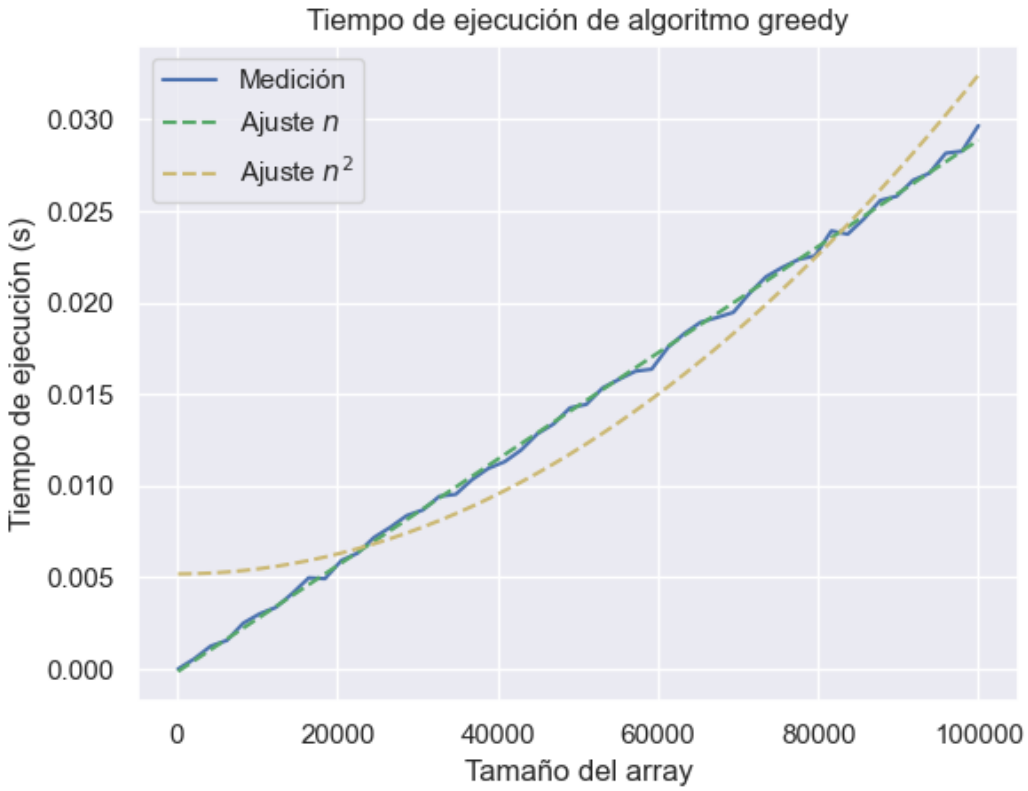

Figura 1: Tiempo de ejecución del algoritmo Greedy y ajustes lineales y cuadráticos. Se puede observar que la medición y el ajuste lineal se asemejan, a diferencia del ajuste cuadrático

Código que pertenece a la sección de errores (usando cuadrados mínimos)

```
errors_n2 = [np.abs(f_n2(n, c_n2[0], c_n2[1]) - results_greedy[n]) for n in x]
errors_n = [np.abs(f_n(n, c_iter[0], c_iter[1]) - results_greedy[n]) for n in x]

print(f"Error cuadratico total para n^2: {np.sum(np.power(errors_n2, 2))}")
print(f"Error cuadratico total para n: {np.sum(np.power(errors_n, 2))}")

ax: plt.Axes
fig, ax = plt.subplots()
ax.plot(x, errors_n2, 'g-',  label="Ajuste $n^2$")
ax.plot(x, errors_n, 'y-', label="Ajuste $n$")
ax.set_title('Error de ajuste')
ax.set_xlabel('Tamano del array')
ax.set_ylabel('Error absoluto (s)')
ax.legend()
None
```

Figura 2: Error cometido entre ajustes.

Como se puede apreciar, el error del ajuste lineal es mas cercano a 0, mientras que el error del ajuste cuadrático es mucho mayor que el anterior. En este caso lo que nos esta indicando el gráfico que la complejidad del algoritmo Greedy no es cuadrática y se asemeja mas a una complejidad lineal.

Lo siguiente es el gráfico que muestra la discrepancia entre una variabilidad alta o baja en los valores de la moneda y el tiempo de ejecución.

Código relevante a esta sección

```
def get_random_array_alta_variabilidad(size: int):
return np.random.randint(1, 100000, size).tolist()
def get_random_array_baja_variabilidad(size: int):
    return np.random.randint(1, 10, size).tolist()
# Siempre seteamos la seed de aleatoridad para que los # resultados sean
    reproducibles
seed(12345)
np.random.seed(12345)
sns.set_theme()

# La variable x van a ser los valores del eje x de los graficos en todo el notebook
# Tamano minimo=100, tamano maximo=100.000 , cantidad de puntos=20
x = np.linspace(100, 100_000, 50).astype(int)

results_greedy_high = time_algorithm(juego_monedas, x, lambda s: [
    get_random_array_alta_variabilidad(s)])
results_greedy_low = time_algorithm(juego_monedas, x, lambda s: [
    get_random_array_baja_variabilidad(s)])

f_n = lambda x, c1, c2: c1 * x + c2

```

```
ax: plt.Axes
fig, ax = plt.subplots()
ax.plot(x, [results_greedy_high[n] for n in x], 'g--', label="Medicion alta
    variabilidad")
ax.plot(x, [results_greedy_low[n] for n in x], 'r--', label="Medicion baja
    variabilidad")

ax.set_title('Tiempo de ejecuciin con diferente variabilidad')
ax.set_xlabel('Tamano del array')
ax.set_ylabel('Tiempo de ejecuciin (s)')
ax.legend()
None

plt.show()
fig.savefig('greedy_variabilidad.png')
```

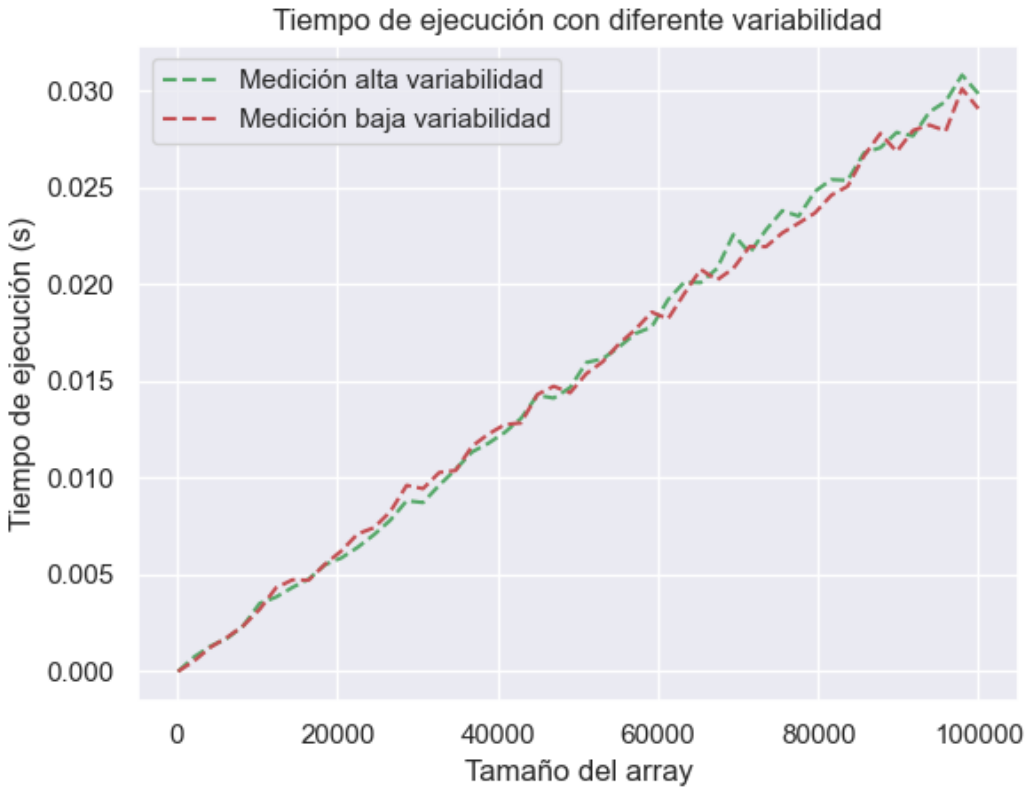

Figura 3: Tiempo de ejecución de variabilidad alta y baja. Se puede observar que el tiempo de ejecución de ambas difiere mínimamente, por lo que se puede concluir que la variabilidad no afecta a los tiempos de ejecución del algoritmo

2. Segunda parte: Mateo empieza a Jugar

En la segunda parte, el juego consiste en el mismo que la primera parte, con la excepción de que Mateo ya puede elegir sus monedas (de forma Greedy) y Sophia empieza a jugar con el uso de un algoritmo en base a un diseño de Programación Dinámica, esto le permite obtener una ventaja al anterior algoritmo y analizaremos este mismo.

2.1. Análisis del problema

Para analizar el problema, describiremos el problema formalmente y de forma mas abstracta. Dado un conjunto de monedas M, definimos dos conjuntos, uno para Sophia y otro para Mateo, cada uno con su respectivo turno.

$$M_i$$
$$M_{m,i}, M_{s,i}$$
$$i : turno; s : Sophia; m : Mateo$$

Para que gane Sophia se tiene que cumplir la misma condición que en la primera parte, siendo

$$\sum_{i=0}^{n} M_{m,i} < \sum_{i=0}^{m} M_{s,i}$$

Para aclaración al lector:

$$n + m = len(M)$$

En cada turno, las opciones de mateo son las siguientes

$$M_0 \; ; \; M_n \; ; \; Con\, n\, siendo\, la\, ultima\, moneda\, del\, conjunto$$

Luego Mateo elegirá la moneda con mayor valor. Esto es importante porque Sophia debe tener en consideración este resultado para poder elegir la mejor opción.

Las opciones de Sophia entonces son:

$$M_0 \; ; \; M_n \; ; \; Con\, n\, siendo\, la\, ultima\, moneda\, del\, conjunto$$

Estas son las opciones que puede elegir, pero la decisión de elegir cada una depende de otras decisiones, ya no es Greedy como en la primera parte. Ahora Sophia debe elegir la moneda que le traerá el mejor resultado, no de forma optima local, si no globalmente. Esta decisión que puede hacer Sophia la vamos a explicar de forma Bottom-Up.

1. Para el caso de n=1 monedas , Sophia elige la moneda restante.
2. Para el caso de n=2, Sophia elige la moneda que mayor valor le suma.
3. Para el caso de n=3 tenemos los siguientes opciones:
   - Sophia elige la primer moneda, Mateo la segunda, Sophia la tercera
   - Sophia elige la primer moneda, Mateo la tercera, Sophia la segunda
   - Sophia elige la tercer moneda, Mateo la primera, Sophia la segunda
   - Sophia elige la tercer moneda, Mateo la segunda, Sophia la primera

En estos 4 casos, Sophia debe tomar la moneda que maximice el valor obtenido y guardarse esa opción. Luego en el siguiente turno de Sophia el problema esta resuelto dado que es un subproblema de largo n=1.

Para un caso de n=k se puede extender, teniendo las siguientes consideraciones

- Sophia elige la moneda n=1 y la mejor opción del subproblema donde obtuvo la moneda n=3
- Sophia elige la moneda n=1 y la mejor opción del subproblema donde obtuvo la moneda n=2
- Sophia elige la moneda n=k y la mejor opción del subproblema donde obtuvo la moneda n=2
- Sophia elige la moneda n=k y la mejor opción del subproblema donde obtuvo la moneda n=1

Luego Sophia debe obtener la opción que le de mas ganancia entre estas cuatro. En el siguiente turno, el problema esta resuelto dado que es un subproblema.

2.2. Análisis de ecuación de recurrencia

$$OPT[i,j] = \text{máx} \begin{cases} \text{monedas}[i] + (OPT[i+2,j], & \text{si mateo elige moneda[i+1])} \\ \text{monedas}[i] + (OPT[i+1,j-1], & \text{si mateo elige moneda[j])} \\ \text{monedas}[j] + (OPT[i,j-2], & \text{si mateo elige moneda[j-1])} \\ \text{monedas}[j] + (OPT[i+1,j-1], & \text{si mateo elige moneda[i])} \end{cases} \quad (1)$$

Parecido a lo que describimos en la opción anterior, OPT[i,j] es la ganancia de Sophia en los pasos anteriores. El indice i indica la moneda que se obtuvo del lado derecho del conjunto de monedas, mientras que j es el indice de la moneda izquierda. Para ejemplificar OPT[i+2,j] representa la mejor opción si se toma la moneda[i+2] siendo que las monedas disponibles esta en el rango [i+2, j]. Esto es, un subproblema diferente a OPT[i+1, j-1], por lo que es necesario una matriz para ir guardando los subproblemas.

Hay que tener en cuenta los casos bases.

Para una cantidad de monedas n = 1, se debe cumplir que

$$OPT[i,j] = monedas[i] \text{si i} = \text{j}$$

Para una cantidad de monedas n = 2, se debe cumplir que

$$OPT[i,j] = max(monedas[i], monedas[j] \text{si i+1} = \text{j}$$

Luego para un caso genérico, es el que describimos anteriormente donde se toma en cuenta las 4 opciones. Para el entendimiento del lector vamos a explicar a que nos referimos con la matriz de OPT. En todos los casos se refiere a la moneda del turno siguiente de Sophia.

- $OPT[i+2,j]$ Mayor puntaje luego de que Sophia elige $moneda[i]$ y Mateo $moneda[i+1]$
- $OPT[i+1,j-1]$ Se divide en dos casos
  - Mayor puntaje luego de que Sophia elige $moneda[i]$ y Mateo $moneda[j]$
  - Mayor puntaje luego de que Sophia elige $moneda[j]$ y Mateo $moneda[i]$
- $OPT[i,j-2]$ Mayor puntaje luego de que Sophia elige $moneda[j]$ y Mateo $moneda[j-1]$

2.3. Algoritmo de programación dinámica propuesto

El siguiente código corresponde a la formación de la matriz OPT

```
def juego_monedas_dinamico(coins):
    n = len(coins)

    opt = [[-1] * n for _ in range(n)]


    for largo_subarreglo in range(1, n + 1):
        for i in range(n - largo_subarreglo + 1):
            j = i + largo_subarreglo - 1

            # Caso base: una sola moneda.
            if i == j:
                opt[i][j] = coins[i]
            # Caso base: dos monedas
            elif i + 1 == j:
                opt[i][j] = max(coins[i], coins[j])
            else:
                # Sophia elige la moneda de la izquierda. Sophia tiene dos opciones
     luego entre i+1;j y
```

```
                # i;j-1. Mateo elige la moneda de mayor valor entre i+1 y j o i y j-1.
                sophia_izq_mateo_izq = coins[i] + opt[i+2][j] if coins[i+1] >= coins[j] and i+2 <= j else 0
                sophia_izq_mateo_der = coins[i] + opt[i+1][j-1] if coins[i+1] < coins[j] and i+1 <= j-1 else 0
                opcion_izq = max(sophia_izq_mateo_izq, sophia_izq_mateo_der)

                # Sophia elige la moneda de la derecha. Sophia tiene dos opciones luego entre i;j-2 y
                # i+1;j-1. Mateo elige la moneda de mayor valor entre i+1 y j o i y j-1.
                sophia_der_mateo_izq = coins[j] + opt[i][j-2] if coins[i] < coins[j-1] and i <= j-2 else 0
                sophia_der_mateo_der = coins[j] + opt[i+1][j-1] if coins[i] >= coins[j-1] and i+1 <= j-1 else 0
                opcion_der = max(sophia_der_mateo_izq, sophia_der_mateo_der)

                opt[i][j] = max(opcion_izq, opcion_der)

    puntaje_sophia = opt[0][n-1]
    puntaje_mateo = sum(coins) - puntaje_sophia

    return puntaje_sophia, puntaje_mateo, reconstruccion(coins, opt)
```

El siguiente código corresponde a la reconstrucción de la solución

```
def reconstruccion(coins, opt):
    elecciones = []
    j = len(coins) - 1
    i = 0

    while i <= j:
        # Obtengo la mejor opcion del lado derecho, la comparo con la mejor opcion.
        # Si no es la misma, avanzo por la otra opcion.
        sophia_der_mateo_izq = coins[j] + opt[i][j-2] if coins[i] < coins[j-1] and i <= j-2 else 0
        sophia_der_mateo_der = coins[j] + opt[i+1][j-1] if coins[i] >= coins[j-1] and i+1 <= j-1 else 0
        opcion_der = max(sophia_der_mateo_izq, sophia_der_mateo_der)

        if opt[i][j] == opcion_der:
            # Se eligio la moneda del lado derecho
            elecciones.append(f"Sophia debe agarrar la ultima ({coins[j]})")
            j -= 1
        else:
            # Se eligio la moneda del lado izquierdo
            elecciones.append(f"Sophia debe agarrar la primera ({coins[i]})")
            i += 1

        if i <= j:
            if coins[i] >= coins[j]:
                elecciones.append(f"Mateo agarra la primera ({coins[i]})")
                i += 1
            else:
                elecciones.append(f"Mateo agarra la ultima ({coins[j]})")
                j -= 1

    return "; ".join(elecciones)
```

2.4. Complejidad

La complejidad del algoritmo en la construcción de OPT es $O(n^2)$ tanto espacial como temporal. Esto se debe a que los subproblemas se guardan en una matriz y hay que recorrerla, sin importar si la mitad de los valores de la matriz no se usan (dado que como máximo Sophia solo puede tomar la mitad de las monedas+1).

En la reconstrucción, la complejidad temporal es $O(n)$ dado que solo se recorre el arreglo de

coins (mondas) y se realizan operacion $O(1)$ como acceder a la matriz OPT o realizar cálculos simples.

2.5. Análisis de optimalidad

Anteriormente explicamos informalmente que se cumple que para un conjunto de monedas n se cumple que se obtiene el mayor puntaje, ya que los subproblemas están resueltos. Vamos a tratar de formalizarlo.

Vimos anteriormente que para los casos base donde n=1,2,3 se obtiene el mayor valor para Sophia. Vamos a tratar de tomar un caso n=j+1, donde para n=j el problema esta resuelto.

Tenemos la matriz OPT(i,j+1)

Si Sophia elige la moneda[i], luego existen dos opciones

- Mateo elige la moneda [i+1], Sophia queda con OPT[i+2, j+1]
- Mateo elige la moneda [j+1], Sophia queda con OPT[i+1, j]

Si Sophia elige la moneda[j+1], luego existen dos opciones

- Mateo elige la moneda [i], Sophia queda con OPT[i+1, j]
- Mateo elige la moneda [j], Sophia queda con OPT[i, j-1]

En los cuatro casos posibles, Sophia queda con un subproblema de largo n=j, donde el subproblema por hipótesis esta resuelto. De esta forma se cumple que la metodología encuentra la solución optima al problema (obtener el mayor puntaje para Sophia).

2.6. Variabilidad

Similar a la primera parte, la variabilidad de los números en las monedas no afecta a los tiempos del algoritmo planteado. Esto sera mostrado con mediciones en la seccion de mediciones.

2.7. Mediciones Segunda Parte

Dataset de prueba

```
def get_random_array(size: int):
    return np.random.randint(1, 1000, size).tolist()
x = np.linspace(100, 2000, 20).astype(int)
results = time_algorithm(juego_monedas_dinamico, x, lambda s: [get_random_array(s)
    ])
```

Gráfico de tiempo según la entrada

```
ax: plt.Axes
fig, ax = plt.subplots()
ax.plot(x, [results[n] for n in x], label="Medicion")
ax.set_title('Tiempo de ejecucion de algoritmo dinamico')
ax.set_xlabel('Tamano del array')
ax.set_ylabel('Tiempo de ejecucion (s)')
ax.legend()
None
```

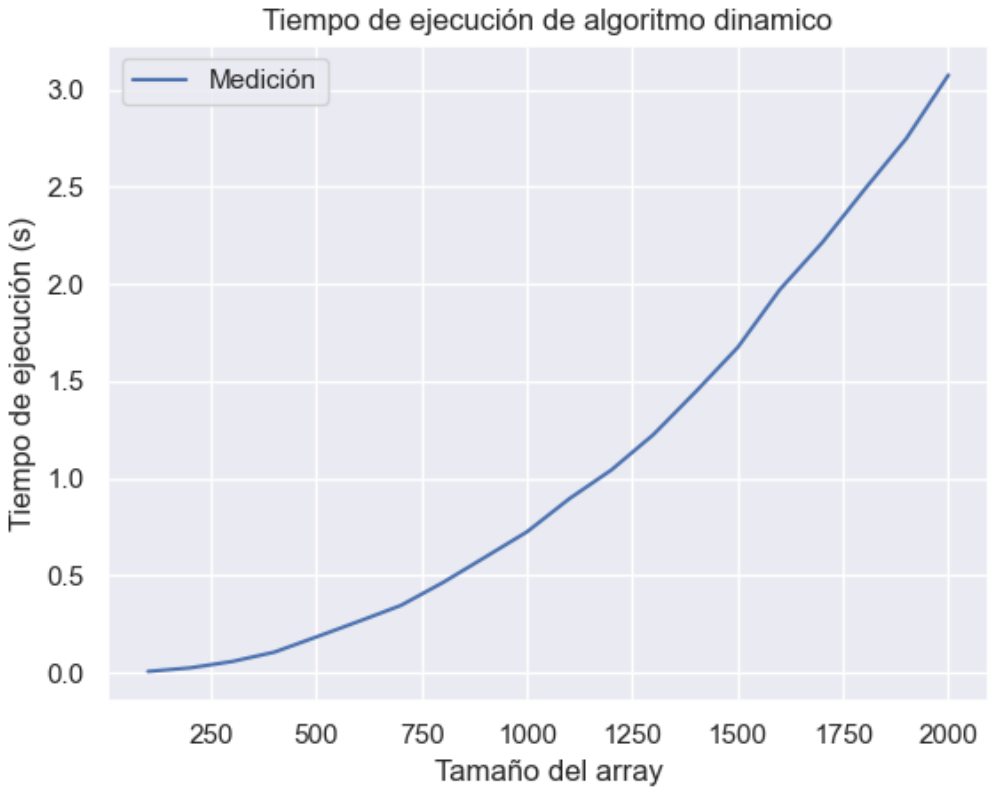

Figura 4: Se observa que para el tamaño 1000 el tiempo que tarda es: 0.75s aprox ,mientras que para el tamaño 2000 el tiempo que tarda es 3s, lo que indica a medida que la entrada aumenta, el tiempo aumenta dramáticamente. Esto se puede notar en la curva que genera.

Gráfico de nuestros datos y el ajuste

```
1
2  f_n = lambda x, c1, c2: c1 * x + c2
3  f_n2 = lambda x, c1, c2: c1 * x**2 + c2
4  f_nlogn = lambda x, c1, c2: c1 * x * np.log(x) + c2
5  c_iter, _ = sp.optimize.curve_fit(f_n, x, [results[n] for n in x])
6  c_n2, _ = sp.optimize.curve_fit(f_n2, x, [results[n] for n in x])
7  c_nlogn, _ = sp.optimize.curve_fit(f_nlogn, x, [results[n] for n in x])
8
9  ax: plt.Axes
10 fig, ax = plt.subplots()
11 ax.plot(x, [results[n] for n in x], label="Medicion")
12 ax.plot(x, [f_n(n, c_iter[0], c_iter[1]) for n in x], 'g--', label="Ajuste $n$")
13 ax.plot(x, [f_n2(n, c_n2[0], c_n2[1]) for n in x], 'y--', label="Ajuste $n^2$")
14 ax.plot(x, [f_nlogn(n, c_nlogn[0], c_nlogn[1]) for n in x], 'r--', label=r"Ajuste
       $n \log(n)$")
15 ax.set_title('Tiempo de ejecucion de algoritmo dinamico con ajustes')
16 ax.set_xlabel('Tamano del array')
17 ax.set_ylabel('Tiempo de ejecucion (s)')
18 ax.legend()
19 None
```

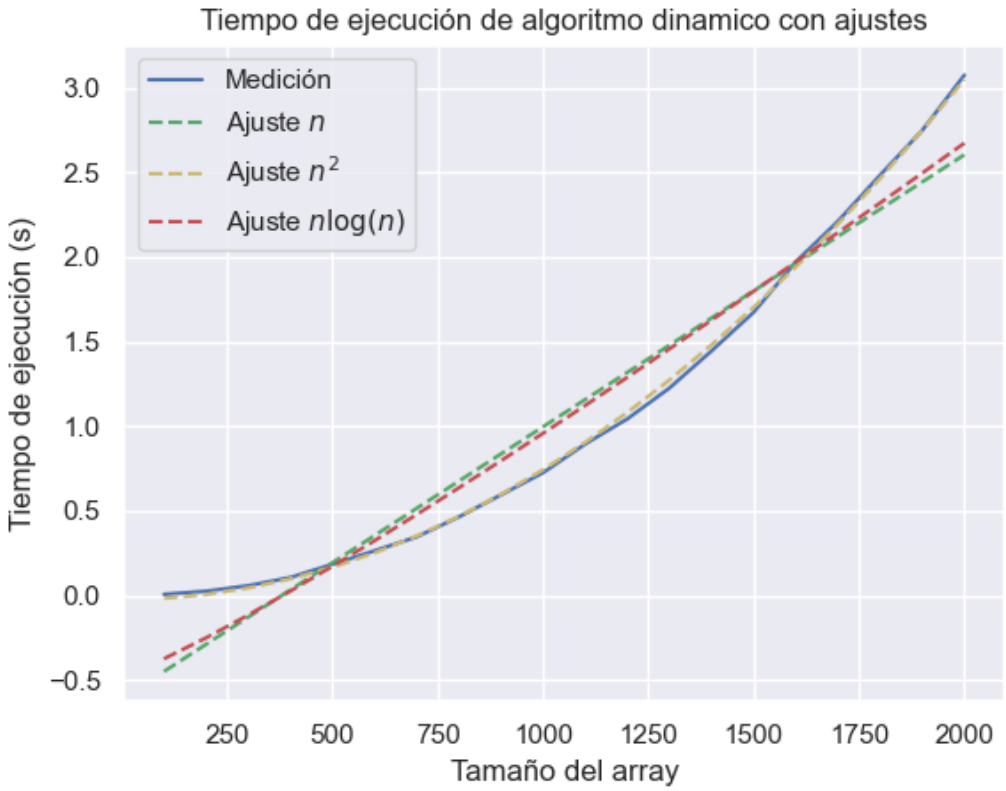

Figura 5: Se puede observar que la medición realizada se alinea con el ajuste cuadrático realizado.

Error cuadrático entre ajustes y el tiempo realizado

```
errors_n2 = [np.abs(f_n2(n, c_n2[0], c_n2[1]) - results[n]) for n in x]
errors_n = [np.abs(f_n(n, c_iter[0], c_iter[1]) - results[n]) for n in x]
errors_nlogn = [np.abs(f_nlogn(n, c_nlogn[0], c_nlogn[1]) - results[n]) for n in x]

print(f"Error cuadratico total para n^2: {np.sum(np.power(errors_n2, 2))}")
print(f"Error cuadratico total para n: {np.sum(np.power(errors_n, 2))}")
print(f"Error cuadratico total para nlogn: {np.sum(np.power(errors_nlogn, 2))}")

ax: plt.Axes
fig, ax = plt.subplots()
ax.plot(x, errors_n2, 'g-',  label="Ajuste $n^2$")
ax.plot(x, errors_n, 'y-', label="Ajuste $n$")
ax.plot(x, errors_nlogn, 'r-', label=r"Ajuste $n \log(n)$")
ax.set_title('Error de ajuste')
ax.set_xlabel('Tamano del array')
ax.set_ylabel('Error absoluto (s)')
ax.legend()
None
```

Error cuadrático total para n^2: 0.010216077623315684
Error cuadrático total para n: 1.1856475484682236
Error cuadrático total para $nlog(n)$: 0.8744827494032212

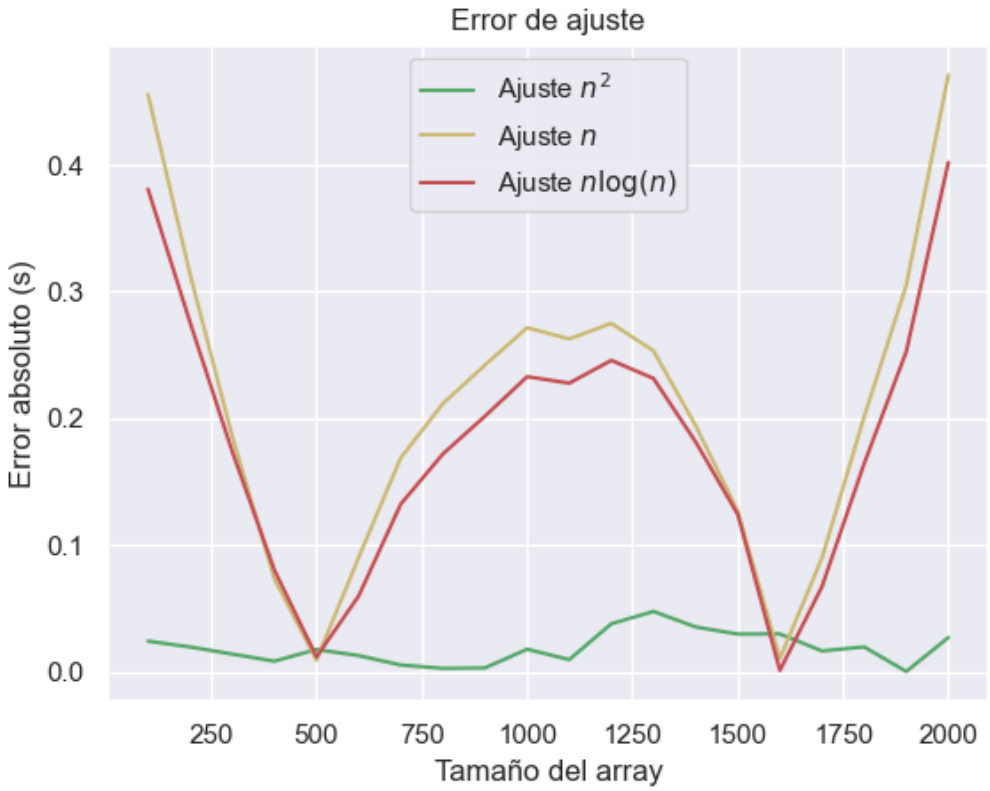

Figura 6: Se puede observar que el error cuadrático medio entre el ajuste cuadrático y el tiempo original es muy próximo a 0, esto indica que el ajuste y la función del tiempo original son muy parecidas, mientras los otros ajustes el error es mucho mayor. Claramente el algoritmo planteado es cuadrático

Tiempo de ejecucion con alta y baja variabilidad

```
def get_random_array_baja_variabilidad(size: int):
    return np.random.randint(1, 100_000, size).tolist()

def get_random_array_alta_variabilidad(size: int):
return np.random.randint(1, 100_000, size).tolist()
```

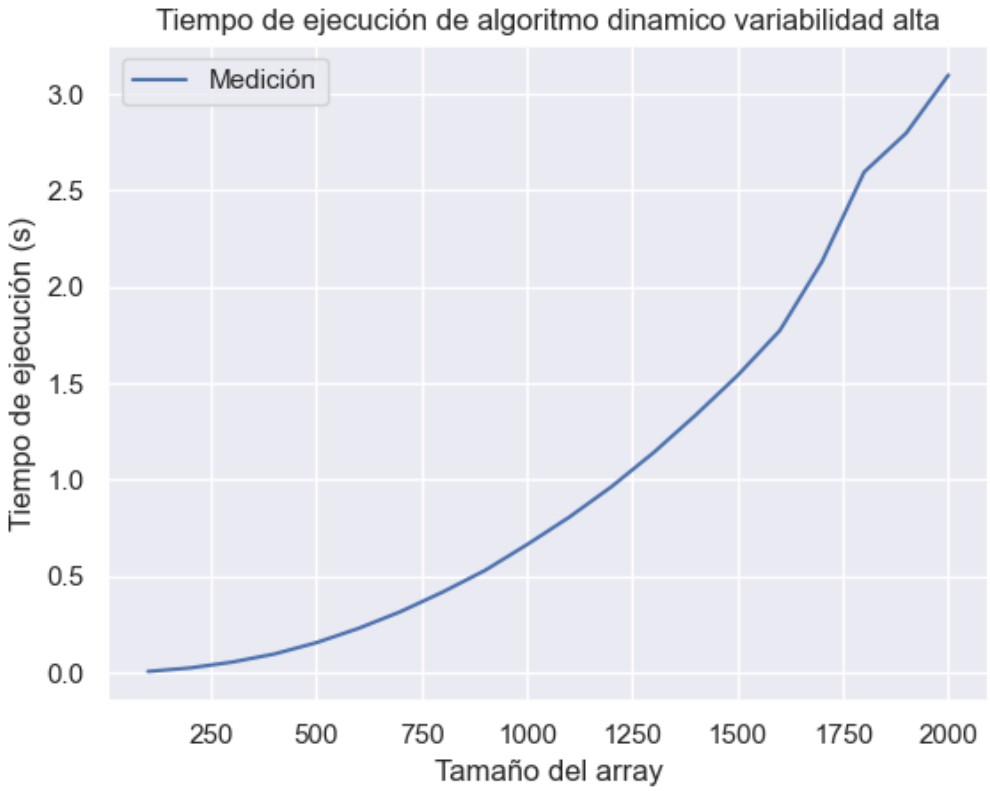

Figura 7: Tiempo de ejecución con alta variabilidad

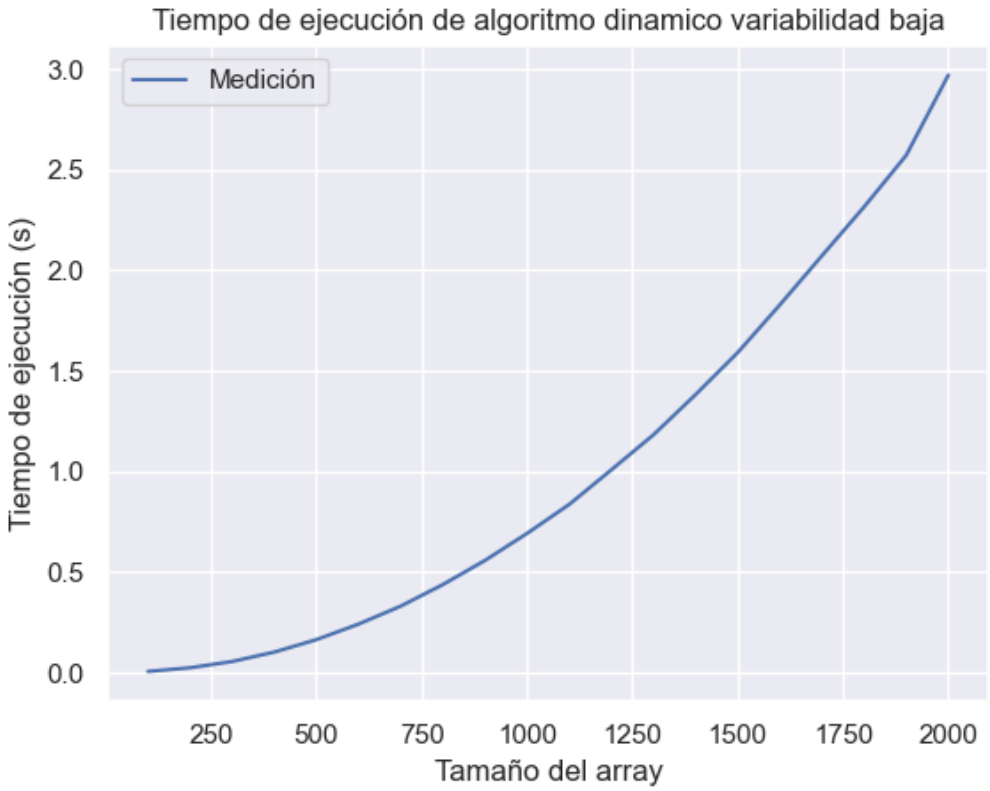

Figura 8: Tiempo de ejecución con alta variabilidad

Como se puede observar, el tiempo de ejecución con ambos, alta y baja variabilidad, no difieren en gran medida. El tiempo de ejecución con alta variabilidad supera aproximadamente 0.1 segundos por lo que se puede observar

3. Tercera parte: Cambios

3.1. Análisis del problema

3.2. Demostración batalla naval es NP

Para asegurar que un problema se encuentra en NP, se requiere de un validador que verifique la validez de la solución propuesta al problema en tiempo polinomial. Para demostrar que el problema de batalla naval se encuentra en NP, se plantea el siguiente algoritmo.

```
def verificar_solucion(tablero, demanda_filas, demanda_columnas):
    for i in range(len(demanda_filas)):
        for j in range(len(demanda_columnas)):
            if tablero[i][j] != 0:
                if not verificar_adyacentes(tablero, i, j):
                    return False
                demanda_filas[i] = demanda_filas[i] - 1
                demanda_columnas[j] = demanda_columnas[j] -1
    for i in range(len(demanda_columnas)):
        if demanda_columnas[i] > 0:
            return False
    for i in range(len(demanda_filas)):
        if demanda_filas[i] > 0:
```

```
            return False

    return True
```

El algoritmo recorre el tablero, y por cada posición que se encuentre ocupado, se van actualizando las demandas correspondientes de cada fila y columna, ademas se verifica que los adyacentes de esa posición no se encuentran ocupados por otro barco. Para verificar los adyacentes se hace uso de la siguiente función.

```
def verificar_adyacentes(tablero, fila, columna):

    for i in range(fila-1,fila+2):
        if i < 0 or i >= len(tablero[0]):
            continue
        for j in range(columna-1,columna+2):
            if j < 0 or j >= len(tablero):
                continue
            if tablero[i][j] != 0 and tablero[i][j] != tablero[fila][columna]:
                return False

    return True
```

3.3. Demostración batalla naval es NP-Completo

3.4. Algoritmo propuesto con backtracking

Se presenta el algoritmo planteado por backtracking del problema de batalla naval individual.

El algoritmo encuentra la solución optima, dado que prueba todas las posibles combinaciones posibles. En otras palabras, prueba colocando un barco en una posición y sin colocarlo, se queda con la mejor solución. Esto se repite en todas las posiciones, sin embargo se pueden realizar podas para que no siga recorriendo ese camino. Esto ultimo lo vamos a detallar en esta sección.

```
def batalla_naval(d_filas, d_columnas, barcos):
    n = len(d_filas)
    m = len(d_columnas)
    tablero = [[0] * m for _ in range(n)]
    barcos.sort(reverse=True)
    solucion = [tablero, float("inf")]
    batalla_naval_bt(tablero, barcos, d_filas, d_columnas, solucion)

    demanda_total = sum(d_filas) + sum(d_columnas)
    demanda_cumplida = demanda_total - solucion[D_INCUMPLIDA]

    return demanda_cumplida, demanda_total, solucion[TABLERO]

def batalla_naval_bt(
    tablero, barcos, d_filas, d_columnas, solucion_parcial
):

    d_incumplida = sum(d_filas) + sum(d_columnas)

    if d_incumplida < solucion_parcial[D_INCUMPLIDA]:
        solucion_parcial[TABLERO] = [list(fila) for fila in tablero]
        solucion_parcial[D_INCUMPLIDA] = d_incumplida

    # Caso base
    if not barcos:
        return
    # Poda si soluci n parcial es peor que la demanda incumplida maxima
    if solucion_parcial[D_INCUMPLIDA] <= d_incumplida - sum(
        barco * 2 for barco in barcos
    ): return

    # Sin barco
    batalla_naval_bt(
        tablero, barcos[1:], d_filas, d_columnas, solucion_parcial
```

```
35      )
36
37      # Con barco
38      for i in range(len(tablero)):
39          if d_filas[i] > 0:  # Hay demanda en la fila?
40              for j in range(len(tablero[0])):
41                  if d_columnas[j] > 0:  # Hay demanda en la columna?
42                      # Pruebo con barco horizontal y vertical
43                      procesar_barco(
44                          tablero,
45                          barcos,
46                          i,
47                          j,
48                          d_filas,
49                          d_columnas,
50                          solucion_parcial,
51                          True,
52                      )
53                      procesar_barco(
54                          tablero,
55                          barcos,
56                          i,
57                          j,
58                          d_filas,
59                          d_columnas,
60                          solucion_parcial,
61                          False,
62                      )
```

En la linea 27 hay una poda, esta corta el recorrido si la demanda de la mejor solución es mejor que la demanda de la solución que se puede llegar a obtener colocando los barcos

```
1  def procesar_barco(
2      tablero,
3      barcos,
4      i,
5      j,
6      d_filas,
7      d_columnas,
8      solucion_parcial,
9      horizontal,
10 ):
11     if intentar_ubicar_barco(
12         tablero, barcos[0], i, j, horizontal, d_filas, d_columnas
13     ):
14         ubicar_barco(
15             tablero, barcos[0], i, j, d_filas, d_columnas, horizontal
16         )
17         batalla_naval_bt(
18             tablero, barcos[1:], d_filas, d_columnas, solucion_parcial
19         )
20         quitar_barco(
21             tablero, barcos[0], i, j, d_filas, d_columnas, horizontal
22         )
```

En esta sección procesamos los barcos, anteriormente iterábamos sobre cada espacio en la matriz llamando a esta función (solo si había demanda en dicha columna) y en esta función intentamos colocar los barcos ya sea de manera horizontal u vertical, llamando a otra función.

```
1  def intentar_ubicar_barco(tablero, barcos, i_fil, i_col, horizontal, d_filas,
       d_columnas):
2      n, m = len(tablero), len(tablero[0])
3
4      # Verificar que el barco no salga del tablero
5      if (horizontal and i_col + barcos > m) or (not horizontal and i_fil + barcos >
       n):
6          return False
7
8      # Verificar que no se violen las demandas de fila y columna
9      if horizontal:
```

```
        if d_filas[i_fil] < barcos:
            return False
        for i in range(barcos):
            if d_columnas[i_col + i] < 1:
                return False
    else:  # Es vertical
        if d_columnas[i_col] < barcos:
            return False
        for i in range(barcos):
            if d_filas[i_fil + i] < 1:
                return False

    # Verificar que no haya barcos adyacentes o superpuestos
    for i in range(barcos):
        fil, col = (i_fil, i_col + i) if horizontal else (i_fil + i, i_col)

        # Revisa que el casillero no est  ocupado por otro barco
        if tablero[fil][col] != 0:
            return False

        # Revisa adyacencias (por fila, columna y diagonales)
        for dx, dy in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0)
    , (1, 1)]:
            adj_fil, adj_col = fil + dx, col + dy
            if 0 <= adj_fil < n and 0 <= adj_col < m and tablero[adj_fil][adj_col]
    != 0:
                return False

    # Verificar extremos del barco
    if horizontal:
        # Solo revisar extremos (izquierda y derecha) una vez
        if (i_col > 0 and tablero[i_fil][i_col - 1] != 0) or (i_col + barcos < m
    and tablero[i_fil][i_col + barcos] != 0):
            return False
    else:  # Es vertical
        # Solo revisar extremos (arriba y abajo) una vez
        if (i_fil > 0 and tablero[i_fil - 1][i_col] != 0) or (i_fil + barcos < n
    and tablero[i_fil + barcos][i_col] != 0):
            return False

    return True
```

Esta función lo que hace es verificar que el barco pueda colocarse, cumpliendo las demandas máximas, que no se salga del tablero, que no tenga barcos adyacentes ni superpuestos.

```
# Ubica al barco en el tablero, de forma horizontal o vertical.
def ubicar_barco(tablero, barcos, i_fil, i_col, d_filas, d_columnas, horizontal):
    if horizontal:
        for i in range(barcos):
            tablero[i_fil][i_col + i] = 1
            d_columnas[i_col + i] -= 1
        d_filas[i_fil] -= barcos
    else:
        for i in range(barcos):
            tablero[i_fil + i][i_col] = 1
            d_filas[i_fil + i] -= 1
        d_columnas[i_col] -= barcos

# Quita al barco del tablero.
def quitar_barco(tablero, barcos, i_fil, i_col, d_filas, d_columnas, horizontal):
    if horizontal:
        for i in range(barcos):
            tablero[i_fil][i_col + i] = 0
            d_columnas[i_col + i] += 1
        d_filas[i_fil] += barcos
    else:
        for i in range(barcos):
            tablero[i_fil + i][i_col] = 0
            d_filas[i_fil + i] += 1
        d_columnas[i_col] += barcos
```

Esta ultima porción de código hace lo que dice hacer, ubica un barco y la otra la quita. Como los barcos están representados por posiciones en el tablero, la lógica es que vaya quitando las posiciones del barco que están adyacentes en el tablero.

El algoritmo, dado que es backtracking, tiene complejidad temporal exponencial.

3.5. Algoritmo propuesto con programación lineal

3.6. Algoritmo propuesto de John Jellicoe

```python
def batalla_naval_greedy(d_filas, d_columnas, barcos):
    n = len(d_filas)
    m = len(d_columnas)
    tablero = [[0] * m for _ in range(n)]
    barcos.sort(reverse=True)

    for barco in barcos:
        ubicado = False

        while not ubicado:
            max_d_fil = max((d, i) for i, d in enumerate(d_filas))
            max_d_col = max((d, j) for j, d in enumerate(d_columnas))

            # El barco es mas grande que la demanda
            if max_d_fil[0] < barco and max_d_col[0] < barco:
                break

            if max_d_fil[0] >= max_d_col[0]:
                # Intenta ubicar por fila
                i_fil = max_d_fil[1]
                for i_col in range(m):
                    #if intentar_ubicar_barco(tablero, m, n, barco, i_fil, i_col, True):
                    if intentar_ubicar_barco(tablero, barco, i_fil, i_col, True, d_filas, d_columnas):
                        ubicar_barco(tablero, barco, i_fil, i_col, d_filas, d_columnas, True)
                        ubicado = True
                        break
            else:
                # Intenta ubicar por columna
                i_col = max_d_col[1]
                for i_fil in range(n):
                    #if intentar_ubicar_barco(tablero, m, n, barco, i_fil, i_col, False):
                    if intentar_ubicar_barco(tablero, barco, i_fil, i_col, False, d_filas, d_columnas):
                        ubicar_barco(tablero, barco, i_fil, i_col, d_filas, d_columnas, False)
                        ubicado = True
                        break

            # No se puede ubicar al barco
            if not ubicado:
                break

    return d_filas, d_columnas, tablero

def intentar_ubicar_barco(tablero, barcos, i_fil, i_col, horizontal, d_filas, d_columnas):
    n, m = len(tablero), len(tablero[0])

    # Verificar que el barco no salga del tablero
    if (horizontal and i_col + barcos > m) or (not horizontal and i_fil + barcos > n):
        return False

    # Verificar que no se violen las demandas de fila y columna
```

```
    if horizontal:
        if d_filas[i_fil] < barcos:
            return False
        for i in range(barcos):
            if d_columnas[i_col + i] < 1:
                return False
    else:  # Es vertical
        if d_columnas[i_col] < barcos:
            return False
        for i in range(barcos):
            if d_filas[i_fil + i] < 1:
                return False

    # Verificar que no haya barcos adyacentes o superpuestos
    for i in range(barcos):
        fil, col = (i_fil, i_col + i) if horizontal else (i_fil + i, i_col)

        # Revisa que el casillero no est  ocupado por otro barco
        if tablero[fil][col] != 0:
            return False

        # Revisa adyacencias (por fila, columna y diagonales)
        for dx, dy in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0)
    , (1, 1)]:
            adj_fil, adj_col = fil + dx, col + dy
            if 0 <= adj_fil < n and 0 <= adj_col < m and tablero[adj_fil][adj_col]
    != 0:
                return False

    # Verificar extremos del barco
    if horizontal:
        # Solo revisar extremos (izquierda y derecha) una vez
        if (i_col > 0 and tablero[i_fil][i_col - 1] != 0) or (i_col + barcos < m
    and tablero[i_fil][i_col + barcos] != 0):
            return False
    else:  # Es vertical
        # Solo revisar extremos (arriba y abajo) una vez
        if (i_fil > 0 and tablero[i_fil - 1][i_col] != 0) or (i_fil + barcos < n
    and tablero[i_fil + barcos][i_col] != 0):
            return False

    return True


# Ubica al barco en el tablero, de forma horizontal o vertical.
def ubicar_barco(tablero, barcos, i_fil, i_col, d_filas, d_columnas, horizontal):
    if horizontal:
        for i in range(barcos):
            tablero[i_fil][i_col + i] = 1
            d_columnas[i_col + i] -= 1
        d_filas[i_fil] -= barcos
    else:
        for i in range(barcos):
            tablero[i_fil + i][i_col] = 1
            d_filas[i_fil + i] -= 1
        d_columnas[i_col] -= barcos

# Quita al barco del tablero.
def quitar_barco(tablero, barcos, i_fil, i_col, d_filas, d_columnas, horizontal):
    if horizontal:
        for i in range(barcos):
            tablero[i_fil][i_col + i] = 0
            d_columnas[i_col + i] += 1
        d_filas[i_fil] += barcos
    else:
        for i in range(barcos):
            tablero[i_fil + i][i_col] = 0
            d_filas[i_fil + i] += 1
        d_columnas[i_col] += barcos

```

```
117
118 def imprimir_juego_naval(tablero):
119     filas = len(tablero)
120     columnas = len(tablero[0]) if filas > 0 else 0
121
122     for i in range(filas):
123         # Imprimir los valores de cada celda en la fila
124         fila = "  ".join("-" if tablero[i][j] == 0 else str(tablero[i][j]) for j in
     range(columnas))
125         print(fila)
```

El algoritmo Greedy de aproximación de batalla naval de John Jellicoe solo difiere en la primera función, el resto las re-uso del algoritmo de backtracking.

Vamos a analizar la complejidad por partes:

1. Ordenamiento de barcos $O(nlogn)$
2. Búsqueda de máxima demanda en filas y columnas $O(f + c)$
3. Ubicar barco en el tablero $O(l(f + c))$ dado que el barco puede tener un largo l

Dado que se repite para cada barco b la complejidad total es $O(b * l * 2(f + c)$. Las funciones de ubicar barco y quitar barco son de complejidad constante $O(1)$

3.7. Mediciones Tercera Parte

En esta sección vamos a analizar las mediciones del algoritmo que resuelve de forma optima la batalla naval usando la técnica de diseño de backtracking con el algoritmo de John Jellicoe, que es una aproximación hacia la resolución optima del problema. A continuación veremos los resultados obtenidos, se muestran los tiempos de los dos algoritmos sobre los datasets utilizados y también la demanda cumplida sobre cada uno de ellos.

Archivo	BT (Duración)	BT (Demanda)	Greedy (Duración)	Greedy (Demanda)
10_10_10.txt	0.088996 s	40	0.000172 s	36
10_3_3.txt	0.000142 s	6	0.000049 s	6
12_12_21.txt	0.888660 s	46	0.000237 s	20
15_10_15.txt	0.053571 s	40	0.000158 s	30
20_20_20.txt	0.874670 s	104	0.000366 s	80
20_25_30.txt	1.597609 s	172	0.000585 s	112
30_25_25.txt	216.040213 s	202	0.000683 s	94
3_3_2.txt	0.000154 s	4	0.000036 s	4
5_5_6.txt	0.018339 s	12	0.000102 s	12
8_7_10.txt	0.012997 s	26	0.000154 s	24

Cuadro 1: Comparación de los algoritmos Backtracking y John Jellicoe en diferentes datasets.

A continuación se muestran los tiempos eliminando el caso del archivo que tomo mas de 3 minutos en ejecutarse, dado que no puede compararse al resto de las pruebas realizadas. Comparado con la segunda prueba que llevo mas tiempo, hay una diferencia de mas de 1000 %Ésto puede por las siguientes razones, una es por falta de podas mas inteligentes y otra es que hay mas demanda y mas espacio, por lo tanto las posibles combinaciones son mucho mayores, y por lo tanto las podas que se encuentran en el código hecho se activan con menos frecuencia.

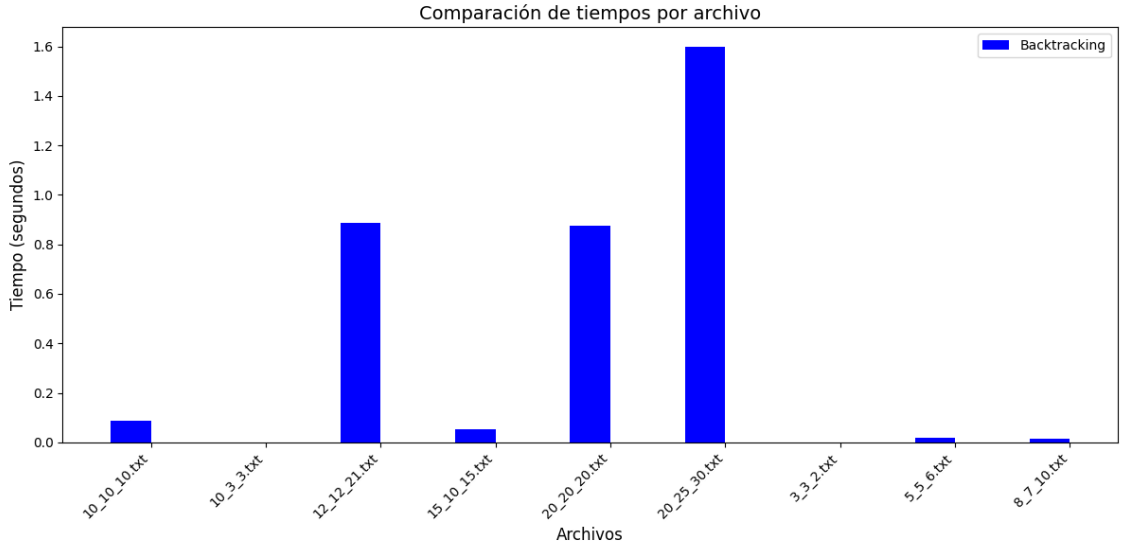

Figura 9: Tiempos de ejecución de los datasets de prueba con el algoritmo optimo de backtracking

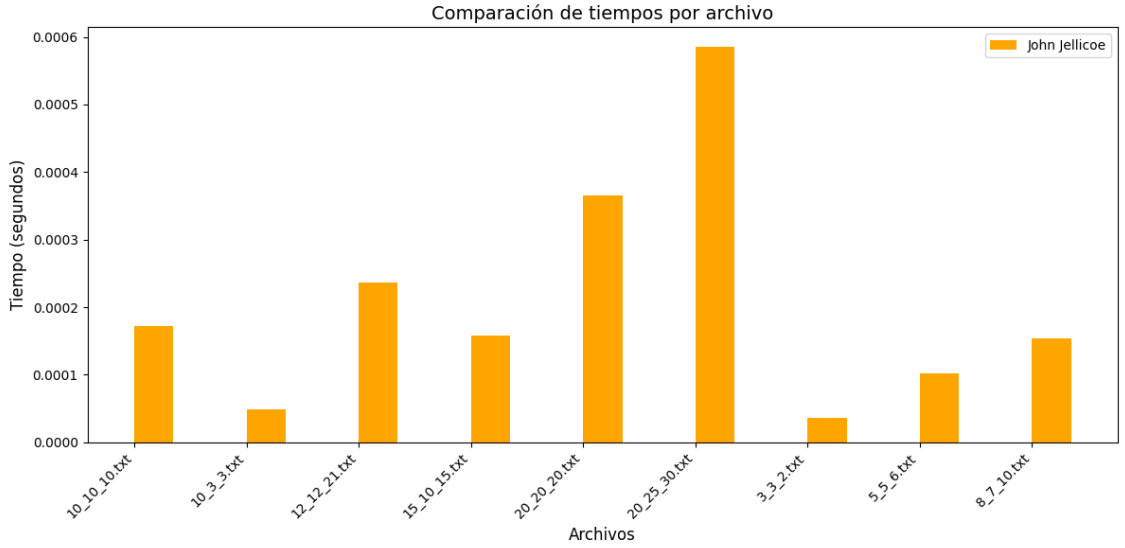

Figura 10: Tiempos de ejecución de los datasets de prueba con el algoritmo Greedy de aproximación

A continuación vamos a calcular a definir la cota que mejor aproxima la solución optima teniendo en cuenta la solución aproximada para cualquier instancia I del problema de la batalla naval. La formula que vamos a utilizar es

$$A(I) \div z(I) \leq r(A)$$

donde $A(I)$ es la solución aproximada y $z(I)$ es la solución optima. En ambos casos usaremos la demanda cumplida.

```
1
2 Archivo: 10_10_10.txt
```

```
Demanda cumplida bt: 40
Demanda cumplida greedy: 36
r(A) >= 0.9000

Archivo: 10_3_3.txt
Demanda cumplida bt: 6
Demanda cumplida: 6
r(A) >= 1.0000

Archivo: 12_12_21.txt
Demanda cumplida bt: 46
Demanda cumplida: 20
r(A) >= 0.4347

Archivo: 15_10_15.txt
Demanda cumplida bt: 40
Demanda cumplida: 30
r(A) >= 0.7500

Archivo: 20_20_20.txt
Demanda cumplida bt: 104
Demanda cumplida: 80
r(A) >= 0.7692

Archivo: 20_25_30.txt
Demanda cumplida bt: 172
Demanda cumplida: 112
r(A) >= 0.6511

Archivo: 30_25_25.txt
Demanda cumplida bt: 202
Demanda cumplida: 94
r(A) >= 0.4653

Archivo: 3_3_2.txt
Demanda cumplida bt: 4
Demanda cumplida: 4
r(A) >= 1.0000

Archivo: 5_5_6.txt
Demanda cumplida bt: 12
Demanda cumplida: 12
r(A) >= 1.0000

Archivo: 8_7_10.txt
Demanda cumplida bt: 26
Demanda cumplida: 24
r(A) >= 0.92307
```

La mínima cota obtenida es en el caso del dataset de prueba 12-12-21 con $r(A) >= 0{,}4347$. Esta cota indica que el algoritmo de John Jellicoe resuelve en un 43 % la demanda optima.

Para corroborar la cota realizamos una prueba con datos lo suficientemente grandes para el algoritmo de backtracking y comparamos con el algoritmo greedy.

Archivo	**BT (Duración)**	**BT (Demanda)**	**Greedy (Duración)**	**Greedy (Demanda)**
30_30_21.txt	8757.204 s	242	0.0005412 s	86

Cuadro 2: Comparación de los algoritmos Backtracking y John Jellicoe con dataset grande.

```
Archivo: 30_30_21.txt
Demanda cumplida bt: 242
Demanda cumplida: 86
r(A) >= 0.35537
```

Haciendo esta comparación, encontramos una nueva cota que sugiere que la aproximación es

menor en esta ocasión.

4. Conclusiones

En conclusión a este trabajo integral, vimos distintas técnicas de diseño de algoritmos bajo los mismos problemas.

- Greedy y Programación dinámica para la primera y segunda parte en el juego de monedas
- Backtracking y aproximaciones en la tercera parte

Entendimos, con la primera y segunda parte, que según el dominio del problema una técnica ya no se podía utilizar, por ejemplo Greedy en la segunda parte no siempre iba a obtener el optimo, y usando programación dinámica en el primero no tenia mucho sentido. En la tercera parte vimos como un algoritmo usando backtracking puede aproximarse con una cota utilizando un algoritmo Greedy, este resultado nos puede ayudar a dar un resultado con cierto margen de error la solución optima a un problema que en muchos casos, puede no realizarse mediante backtracking por la cantidad de tiempo y procesamiento que puede realizar.

Por otra parte a través de este trabajo practico utilizamos herramientas para realizar diferentes pruebas y aproximaciones, corroboramos las complejidades temporales de ejecución de los algoritmos de la primera y segunda parte, comprobando que la complejidad analizada de los algoritmos se asemejaba a la analizada.