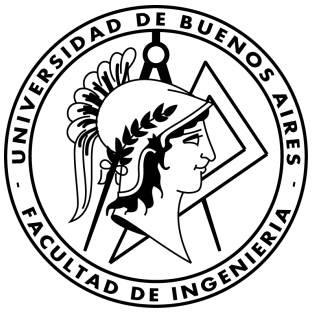


Trabajo Práctico — Juego de Hermanos

[7541/9515] Teoria de Algoritmos
Segundo cuatrimestre de 2024

Alumno	Número de padrón	Email
Venglar, Matias	103545	mvenglar@fi.uba.ar
De Nobili, Lautaro	107394	ldenobili@fi.uba.ar
Alzogaray, Joaquin	108905	jalzogaray@fi.uba.ar

Índice

1. Introducción

1.1. Consigna general y objetivo.

El trabajo práctico se encuentra dividido en tres partes en las cuales se exploran y desarrollan distintos algoritmos, utilizando como temática a dos hermanos, Sophia y Mateo, que juegan y aprenden a través de estos algoritmos.

En la primera parte, se implementa un algoritmo Greedy para resolver el juego con n Monedas, con las respectivas restricciones que tienen Sophia y Mateo para jugar. En la segunda parte, se utiliza Programación Dinámica, una nueva herramienta adquirida por Sophia, para jugar con n Monedas, pero está vez con Mateo haciendo uso de Greedy. Por último, en la tercera parte, se implementan algoritmos de Backtracking, Programación Lineal, y Greedy para resolver el juego "La Batalla Naval Individual", creado en Argentina.

El objetivo en todas partes es encontrar la solución óptima al problema planteado, utilizando los algoritmos especificados. Además, se deben de realizar sus respectivos análisis de complejidad y mediciones de tiempo, para medir y comparar entre si.

2. Teoría sobre los Algoritmos utilizados.

En este apartado, se expondran las diferentes definiciones teóricas de los distintos Algoritmos que se utilizaran para resolver los problemas planteados en la consigna del trabajo práctico.

2.1. ¿Qué es un Algoritmo Greedy?

Un Algoritmo Greedy (o codicioso) es una técnica de programación que consiste en aplicar una regla sencilla que nos permita obtener el óptimo local para el estado actual del problema en cuestión. Esta regla se aplica iterativamente con la finalidad de obtener una solución óptima global.

Características de los Algoritmos Greedy:

- eligen la mejor opción para el estado actual, esperando que estas decisiones conduzcan a una solución global óptima;
- una vez que se toma una decisión, no se revisan decisiones previas. Es decir, Greedy no se arrepiente;
- son intuitivos de pensar, fáciles de entender y plantear, y suelen funcionar con una baja complejidad algorítmica.

2.2. ¿Qué es un Algoritmo por Backtracking?

Un algoritmo de Backtracking es una técnica de programación que implica explorar sistemáticamente todas las posibles soluciones de un problema mediante una búsqueda en profundidad, y retroceder o podar (o hacer "backtrack") cuando se determina que la solución actual no puede llevar a una solución completa, válida y/o mejor que la mejor solución probada hasta el momento.

El proceso general del Backtracking incluye:

- se toma un elemento del problema como contenido en la mejor solución, y se continúa con el siguiente elemento;
- si se determina que la opción elegida no puede conducir a una solución completa, válida y/o mejor, se retrocede (backtrack) al estado anterior y se elige el siguiente elemento como contenido en la mejor solución, repitiendo el ítem anterior;
- si se encuentra una solución completa, válida y mejor, se la guarda o se maneja, según los requisitos del problema.

2.3. ¿Qué es la Programación Dinámica?

La Programación Dinámica es una técnica de programación que resuelve problemas de optimización dividiéndolos en subproblemas más pequeños, resolviendo cada subproblema una vez y almacenando sus soluciones para poder reutilizarlas y combinarlas para resolver el problema más grande. Está técnica de almacenar los resultados previamente calculados, así no volver a calcularlos, se llama Memoization.

La resolución de un problema mediante Programación Dinámica implica:

- se definen los subproblemas, así un problema grande se divide en pequeños subproblemas;
- se resuelve el subproblema identificado utilizando iteraciones o recursividad, prefiriendo la primera para hacer mejor uso del stack memoria del computador;
- las soluciones a los subproblemas se almacenan para poder ser reutilizadas;
- la solución al problema principal se construye a partir de los subproblemas que ya se han calculado.

2.4. ¿Qué es un Modelo de Programación Lineal?

Programación Lineal es una técnica de diseño matemática que permite resolver problemas de optimización de un sistema de ecuaciones lineal con definidas variables y restricciones que han de cumplir las mismas.

Para resolver un problema por Programación Linea, se debe de:

- plantear variables, sean contínuas o enteras, según requiera el problema;
- plantear ecuaciones e inecuaciones lineales que definen restricciones sobre esas variables;
- definir la función objetivo que buscamos maximizar o minimizar;
- tener un modelo que resuelva dicho sistema por Programación Lineal, como es PuLP, que utiliza el método Simplex.

2.5. ¿Qué es NP y NP-Completo?

En la teoría de la complejidad computacional, los problemas se clasifican en diversas clases en función de su capacidad de resolución eficiente. El conjunto P abarca problemas que pueden resolverse de manera eficiente en **tiempo polinomial** mediante una máquina de Turing determinística. Por otro lado, la clase NP engloba problemas para los cuales existe un **certificador eficiente**, lo que implica que la solución puede ser validada en tiempo polinomial. Es decir, estos problemas pueden resolverse en tiempo polinomial por una máquina de Turing no determinística. Un problema NP-completo es un tipo especial de problema en la teoría de la complejidad computacional. Un problema específico es **NP-completo** si pertenece a la clase NP (donde las soluciones pueden ser verificadas eficientemente en tiempo polinomial), y al mismo tiempo, cualquier problema en la clase NP puede reducirse de manera polinomial a él. En otras palabras, un problema NP es tan difícil como lo es un problema NP-completo, al cual se puede reducir,

2.6. ¿Qué es un Algoritmo por Aproximación?

Estos algoritmos surgen de la pregunta ¿cómo diseñamos algoritmos para problemas en los que el tiempo polinomial es probablemente inconseguible?

Características:

- diseñados para ejecutarse en tiempo polinomial;
- encuentran soluciones que garantizan estar cerca del óptimo;

- se puede demostrar que existe una solución cerca del óptimo;
- la solución óptima es muy difícil de calcular y aún así es posible demostrar la cercanía de la solución aproximada.

Hay distintas metodologías para implementar Algoritmos de Aproximación: Greedy, Pricing Method, Programación Lineal, Redondeos, Programación Entera y Programación Dinámica.

3. Parte 1: Juego de Monedas usando un Algoritmo Greedy.

El juego consiste en: se dispone una fila de n monedas, de diferentes valores. En cada turno, un jugador debe elegir alguna moneda, pero no puede elegir cualquiera: sólo puede elegir o bien la primera de la fila, o bien la última. Al elegirla, la remueve de la fila, y le toca luego al otro jugador, quien debe elegir otra moneda siguiendo la misma regla. Ambos jugadores siguen agarrando monedas hasta que no quede ninguna en la fila. El ganador será quien tenga el mayor valor acumulado (por sumatoria).

3.1. Análisis del Problema.

Se pide implementar un Algoritmo Greedy que encuentre la solución óptima al problema: Sophia debe ganar siempre.

Para esto, la regla Greedy elegida es que Sophia, en cada turno para ella, siempre elija la moneda que más valor tiene entre ambas opciones; mientras que Mateo, "jugando"mediante su hermana, en cada turno siempre debe elegir la moneda de menos valor entre ambas opciones. De esta forma, la solución óptima global para Sophia se asegura por dos vías: siempre elije como óptimo local la moneda de mayor valor para maximizar su sumatoria parcial y, además, siempre elije como "pésimo local"la moneda de menor valor para Mateo de forma que siempre se asegura de tener la moneda de mayor valor para su próximo turno.

La complejidad del Algoritmo Greedy planteado se puede analizar de la siguiente manera:

- recorrer todos los datos una vez en el arreglo tiene una complejidad O(n);
- acceder cada vez al arreglo, utilizar la función append y el condicional if tiene una complejidad O(1);
- cada adición a la sumatoria de valor de cada jugador tiene una complejidad O(n/2) parcial y O(n) total.

Por lo tanto, en total la complejidad del algoritmo es O(n).

Adicionalmente, se puede afirmar que, la variabilidad de los valores de las diferentes monedas a los tiempos del algoritmo planteado no afectan de ningún modo debido a que las comparaciones siempre son directas entre los valores de las dos monedas para cada turno y esta comparación siempre se encuentra en tiempo O(1). Por otro lado, como ya fue planteado también anteriormente, la variabilidad de los valores de las diferentes monedas no afecta a la optimalidad del algoritmo planteado. Esto es así porque en cada turno, Sophia siempre se asegura de elegir la moneda de mayor valor para ella misma y se asegura de elegir la moneda de menor valor para Mateo, lo que le da la posibilidad de que siempre le quede una moneda de mayor valor para su próximo turno y, en caso de que sea una cantidad par de monedas y se encuentren en la última ronda, primeramente siempre hubiera elegido la moneda de mayor valor para ella misma por busqueda del óptimo local "propio". Por último, como Sophia siempre juega primero, la cantidad de monedas que obtiene siempre será mayor o igual a las de Mateo.

3.2. Implementación del Algoritmo.

El algoritmo, entonces, es un algoritmo Greedy simple y fácil de entender. A través de un ciclo while se comparan la primer y última moneda por turno y, en base a dicha comparación, se "toma"la moneda de mayor valor siempre para Sophia y la moneda de menor valor siempre para Mateo, como se planteó con antelación.

```
def greedy(arr_datos):
    turnos = []
    monedas_sophia = []
    monedas_mateo = []

    i_primera = 0
    i_ultima = len(arr_datos) - 1
    turno_sophia = True
    while i_primera <= i_ultima:
        if turno_sophia:
            if arr_datos[i_primera] >= arr_datos[i_ultima]:
                turno("Primera moneda para Sophia", turnos, i_primera, monedas_sophia, arr_datos)
                i_primera += 1
            else:
                turno("Última moneda para Sophia", turnos, i_ultima, monedas_sophia, arr_datos)
                i_ultima -= 1
        else:
            if arr_datos[i_primera] <= arr_datos[i_ultima]:
                turno("Primera moneda para Mateo", turnos, i_primera, monedas_mateo, arr_datos)
                i_primera += 1
            else:
                turno("Última moneda para Mateo", turnos, i_ultima, monedas_mateo, arr_datos)
                i_ultima -= 1
        turno_sophia = not turno_sophia

    return [turnos, "Ganancia de Sophia: " + str(sum(monedas_sophia))]
```

3.3. Mediciones.

Se presentan las mediciones hechas para corroborar la complejidad teórica indicada más arriba: la complejidad del algoritmo es O(n). Esta corroboración empírica, a través de diferentes casos de prueba, se realizo utilizando la técnica de cuadrados mínimos. Los arreglos utilizados fueron de 20 a 20000 Monedas cada uno.

Por un lado, se tiene el tiempo de ejecución del algoritmo:

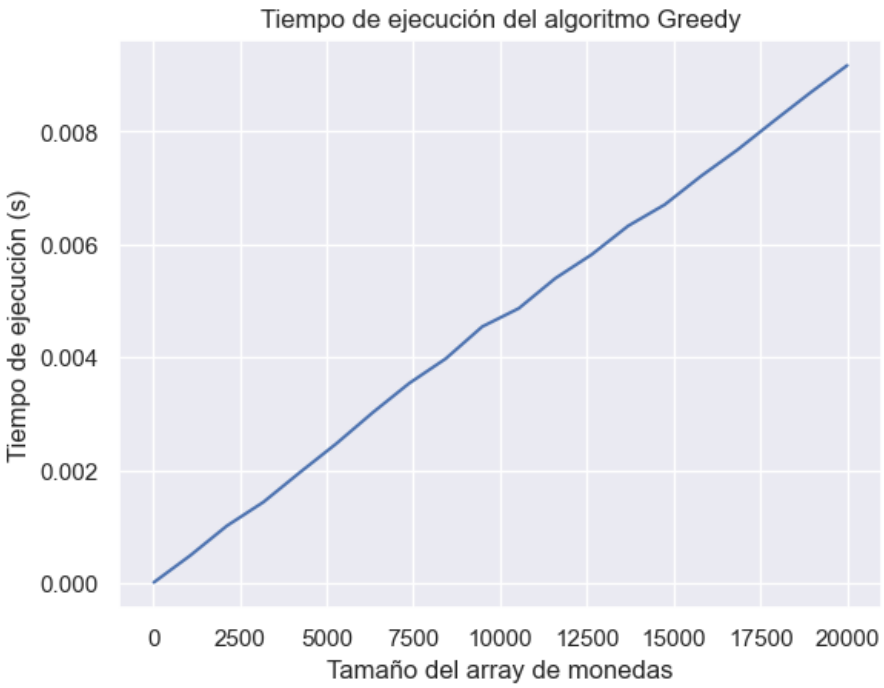


Figura 1: Tiempo de ejecución del Algoritmo Greedy.

Por otro lado, en el siguiente gráfico se ajusta una recta a la ejecución del algoritmo y se puede observar que la recta aproxima muy bien al tiempo de ejecución.

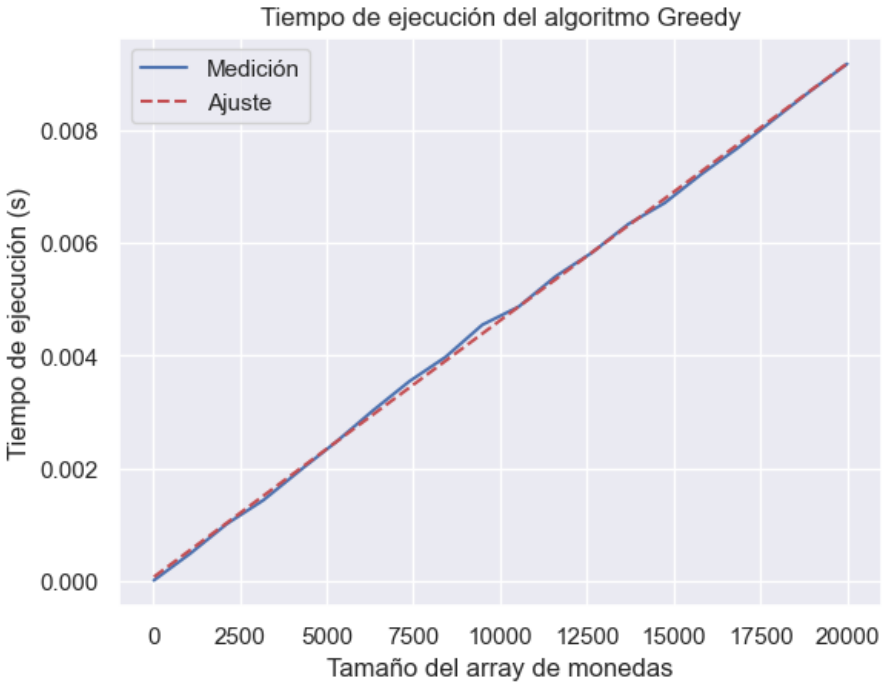


Figura 2: Ajuste del Algoritmo Greedy.

Si calculamos el error cuadrático, se puede observar que el error del ajuste es pequeño, por lo

que se puede afirmar que la complejidad del algoritmo es O(n).

Figura 3: Error Cuadrático del Algoritmo Greedy.

3.4. Conclusiones.

Para concluir, se puede afirmar que, por la forma que está pensado el problema, un algoritmo Greedy es una técnica fácil, intuitiva y rápida para encontrar una solución. Además, en este caso se puede comprobar que el algoritmo propuesto, además de ser una solución óptima para el problema, gracias a los óptimos locales propuestos para cada turno, también es una solución que tiene una complejidad lineal u O(n), lo que lo convierte en una solución eficiente temporalmente.

4. Parte 2: Juego de Monedas usando Programación Dinámica.

Pasaron los años y los hermanos crecieron. Ahora, Mateo aprendió a jugar al Juego de Monedas aplicando la técnica Greedy que utilizaba su hermana de chica, pero como Sophia también creció, también fue adquiriendo nuevas herramientas: ahora aprendió sobre Programación Dinámica y va a utilizar esta técnica para asegurarse de ganar siempre que pueda.

4.1. Análisis del Problema.

La consigna pide utilizar Programación Dinámica para resolver el problema de maximizar la ganancia de Sophia, mientras Mateo juega utilizando un enfoque Greedy. En este caso, para cada turno, Sophia tendra que pensar un paso adelante para tomar mejores decisiones y así ganar siempre que pueda. Entonces, para cada vez que le toque elegir una moneda debera pensar en cuatro opciones:

- tomar la primer moneda, sabiendo que Mateo en su turno tomará la primera por ser la mayor entre ella y la última;

- tomar la primer moneda, sabiendo que Mateo en su turno tomará la última por ser la mayor entre ella y la primera;
- tomar la última moneda, sabiendo que Mateo en su turno tomará la primera por ser la mayor entre ella y la última;
- tomar la última moneda, sabiendo que Mateo en su turno tomará la última por ser la mayor entre ella y la primera.

Para cada caso, ella escogerá la opción que maximiza su ganancia final y minimiza la de Mateo, que no siempre será la de mayor valor en el turno, sino la que le asegure que al final del juego tenga la mayor sumatoria posible. En cambio, Mateo siempre elegira la de mayor valor en cada turno.

4.2. La ecuación de recurrencia lleva al máximo valor acumulado posible.

Para demostrar que la ecuación de recurrencia formulada lleva al máximo valor acumulado posible, se utiliza el método de indución matemática, planteando primero los casos base:

1. $i = f -> OPT(i, i) = V[i]$ que es óptimo porque es la unica moneda del arreglo, y por lo tanto el máximo valor;
2. $i = f - 1 -> OPT(i, i + 1) = max(V[i], V[i + 1])$ que es óptimo porque se toma la moneda de mayor valor entre las dos posibles.

Luego se continua con el paso inductivo, en donde se repite, con llamados recursivos, la ecuación de recurrencia hasta lograr llegar a uno de los casos base, partiendo de la ecuación de recurrencia:

$$OPT(i, f) =$$

$$max(V[i] + (OPT(i + 1, f - 1) si V[i + 1] < V[f] sino OPT(i + 2, f)),$$
$$V[f] + (OPT(i + 1, f - 1) si V[i] > V[f - 1] sino OPT(i, f - 2)))$$

Siendo:

- i = indice primer valor del arreglo;
- f = indice ultimo valor del arreglo (len(V) - 1);
- max() = porque Sophia quiere maximizar su ganancia.
- V[i] = primer valor del arreglo, si lo toma, se pasa a (i + 1, f);
- V[j] = ultimo valor del arreglo, si lo toma, se pasa a (i, f - 1);
- OPT(i + 1, f - 1) = implica que Mateo se agarro el ultimo, V[f], posterior a Sophia agarrar el primero, V[i];
- OPT(i + 2, f) = implica que Mateo se agarro el primero, V[i], posterior a Sophia agarrar el primero, V[i];
- OPT(i + 1, f - 1) = implica que Mateo se agarro el primero, V[i], posterior a Sophia agarrar el ultimo, V[f];
- OPT(i, f - 2) = implica que Mateo se agarro el ultimo, V[f], posterior a Sophia agarrar el ultimo, V[i].

Entonces, se tiene así tres posibilidades para los parámetros del llamado recursivo:

1. OPT(i+1,f-1);
2. OPT(i+2,f);
3. OPT(i,f-2).

Las cuales corresponden a los siguientes casos:

	Sophia agarra el primero	Sophia agarra el ultimo
Mateo agarra el primero	OPT(i + 2, f)	OPT(i + 1, f - 1)
Mateo agarra el ultimo	OPT(i + 1, f - 1)	OPT(i, f - 2)

Suponiendo que estos casos son óptimos, entonces OPT(i,f) es óptimo como consecuencia de Sophia prediciendo el comportamiento de Mateo como se explico anteriormente, quedando así demostrado OPT(i,f) como óptimo.

4.3. Implementación del Algoritmo.

```python
def pdinamica(monedas):
    cant_monedas = len(monedas)
    tabla = [[0 for _ in range(cant_monedas)] for _ in range(cant_monedas)]

    for indice in range(cant_monedas):
        tabla[indice][indice] = monedas[indice]
        if indice < cant_monedas - 1:
            tabla[indice][indice + 1] = max(monedas[indice], monedas[indice + 1])

    for j in range(2, cant_monedas):
        for i in range(cant_monedas - j):
            f = i + j
            tabla[i][f] = max(
                monedas[i] + (tabla[i + 1][f - 1]
                if monedas[i + 1] < monedas[f] else tabla[i + 2][f]),
                monedas[f] + (tabla[i + 1][f - 1]
                if monedas[i] > monedas[f - 1] else tabla[i][f - 2]))

    return reconstruccion(monedas, tabla)
```

```python
def reconstruccion(monedas, tabla):
    turnos = []
    monedas_sophia = []
    monedas_mateo = []
    i = 0
    f = len(monedas) - 1
    turno_sophia = True

    while f >= i:
        if turno_sophia:
            i_aux_primera_moneda = (i + 1) if monedas[i + 1] < monedas[f] else (i + 2)
            f_aux_primera_moneda = (f - 1) if monedas[i + 1] < monedas[f] else f
            i_aux_ultima_moneda = (i + 1) if monedas[i] > monedas[f - 1] else i
            f_aux_ultima_moneda = (f - 1) if monedas[i] > monedas[f - 1] else (f - 2)

            if (monedas[f] + tabla[i_aux_ultima_moneda][f_aux_ultima_moneda] >
                    monedas[i] + tabla[i_aux_primera_moneda][f_aux_primera_moneda]):
```

```
                turno("Última moneda para Sophia", turnos, f, monedas_sophia, monedas)
                f -= 1
            else:
                turno("Primera moneda para Sophia", turnos, i, monedas_sophia, monedas)
                i += 1

        else:
            if monedas[f] >= monedas[i]:
                turno("Última moneda para Mateo", turnos, f, monedas_mateo, monedas)
                f -= 1
            else:
                turno("Primera moneda para Mateo", turnos, i, monedas_mateo, monedas)
                i += 1

        turno_sophia = not turno_sophia

    return [turnos,
            "Ganancia Sophia: " + str(sum(monedas_sophia)),
            "Ganancia Mateo: " + str(sum(monedas_mateo))
            ]
```

4.4. Mediciones.

Se presentan las mediciones hechas para corroborar la complejidad teórica indicada más arriba: la complejidad del algoritmo es $O(n^2)$. Esta corroboración empírica, a través de diferentes casos de prueba, se realizo utilizando la técnica de cuadrados mínimos. Los arreglos utilizados fueron de 20 a 1000 Monedas cada uno.

Por un lado, se tiene el tiempo de ejecución del algoritmo:

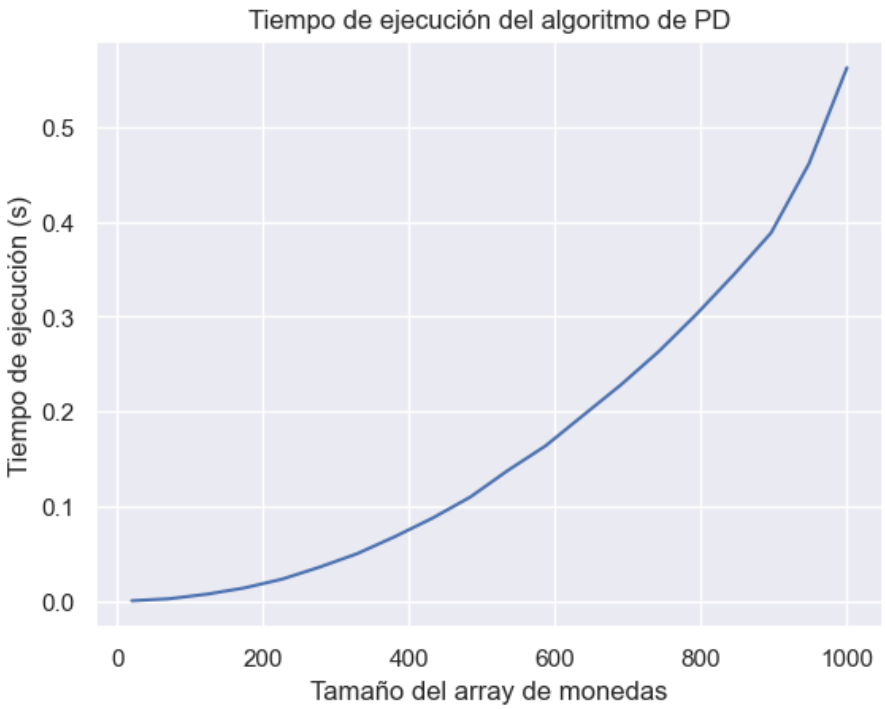


Figura 4: Tiempo de ejecución del Algoritmo PD.

Por otro lado, en el siguiente gráfico se ajusta una curva cuadrática a la ejecución del algoritmo y se puede observar que la curva aproxima muy bien al tiempo de ejecución.

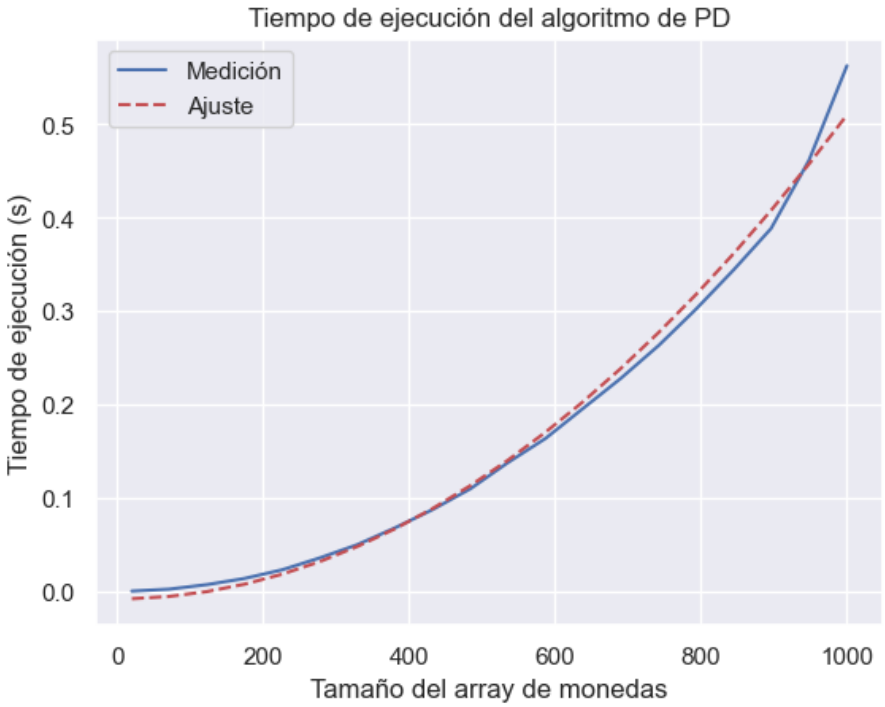


Figura 5: Ajuste del Algoritmo PD.

Si calculamos el error cuadrático, se puede observar que el error del ajuste es pequeño, por lo que se puede afirmar que la complejidad del algoritmo es $O(n^2)$.

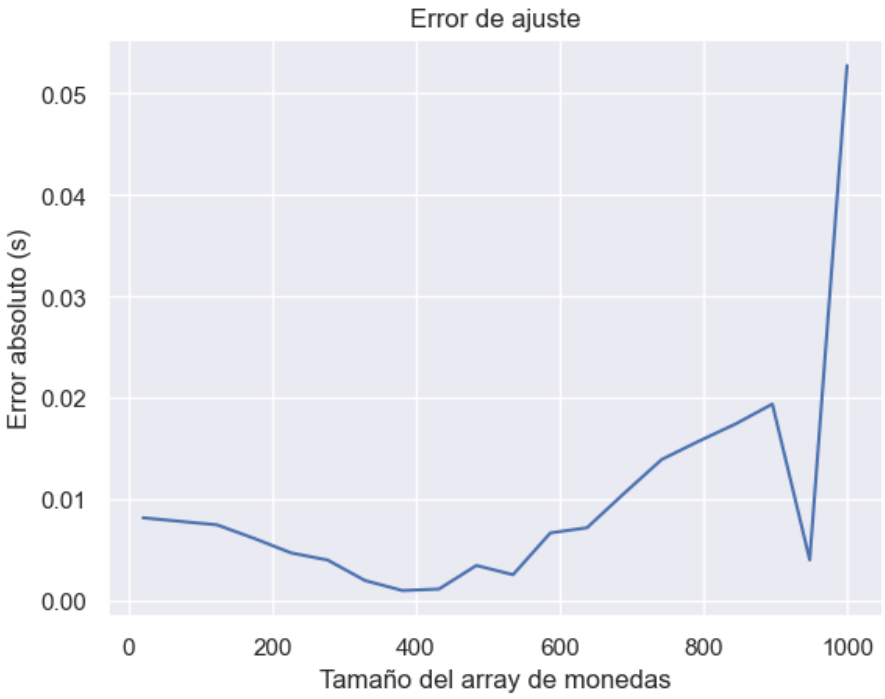


Figura 6: Error Cuadrático del Algoritmo PD.

4.5. Conclusiones.

Al suponer que la solución óptima al problema se construye junto con la solución óptima de una versión reducida de dicho problema, se logra un algoritmo elegante el cual, iterativamente, se rellena un arreglo de datos útil para realizarse una reconstrucción de los pasos que siguen Sophia y Mateo en su juego. No solo se hace uso del supuesto de ɀa se poseé la solución para este problema reducido", sino que, al usar Memoization, se maneja eficientemente la memoria del stack, y la reconstrucción se realiza con una baja complejidad temporal, sin mencionar que siempre se logra el máximo valor posible acumulado para Sophia. Si bien esta técnica de programación puede resultar compleja de entender e implementar cuando uno recién adquiere dicho conocimiento, una vez que se analiza con todos los ejemplos brindados por la cátedra y se realizan seguimientos con varios ejemplos para la ecuación de recurrencia, no solo se logra identificar posibles fallas en dicha ecuación, sino que se desbloquea un nuevo mundo de entendimiento para esta tan elegante técnica de programación.

5. Parte 3: La Batalla Naval Individual.

Los hermanos siguieron creciendo. Mateo también aprendió sobre Programación Dinámica y cada uno aplicaba la lógica sabiendo que el otro también lo hacía. El juego de las monedas se tornó aburrido en cuánto notaron que siempre ganaba quien empezara, o según la suerte. Los años pasaron, llegó la adolescencia y empezaron a tener gustos diferentes. En general, jugaban a juegos individuales. En particular, Sophia estaba muy enganchada con un juego inventado en Argentina por Jaime Poniachik (uno de los fundadores de Ediciones de Mente) en 1982: La Batalla Naval Individual.
En dicho juego, tenemos un tablero de n x m casilleros, y k barcos. Cada barco i tiene bi casilleros

de largo. Es decir, requiere de *bi* casilleros para ser ubicado. Todos los barcos tienen 1 casillero de ancho. El tablero a su vez tiene un requisito de consumo tanto en sus filas como en sus columnas. Si en una fila indica un 3, significa que deben haber 3 casilleros de dicha fila siendo ocupados. Ni más, ni menos. No podemos poner dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente).

5.1. El Problema de la Batalla Naval se encuentra en NP.

Para esta demostración se debe proponer un certificador que valide su resultado en tiempo polinomial (no tiene que resolverlo, solo validar lo que devolvio). En este caso, el problema devuelve un tablero que, para validarlo, se pueden recorrer todas sus casillas con una complejidad O (n x m), disminuyendo en 1 los valores de demanda, en sus respectivos indices, en las listas de fila y columna, si la casilla que se está recorriendo esta ocupada. Terminado el recorrido de todo el tablero, se hace la sumatoria de ambas listas. Esto sería el resultado de las demandas de fila y columna insatisfechas, en O (n) y O (m) respectivamente. Si la cantidad de demanda insatisfecha es 0, entonces esto valida que la solución es válida, caso contrario no lo es. La previa demostración no es exclusiva, implicando que pueden haber otros validadores (y así demostraciones) para el Problema de la Batalla Naval.

5.2. El Problema de la Batalla Naval es NP-Completo.

Para esta demostración se necesita reducir un problema Y $NP-Completo$ que se pueda demostrar que pertenece a este conjunto de problemas, al problema X de *La Batalla Naval*. Entonces, se propone reducir el problema de Independent Set, habiendo demostrado su NP-Completitud en clase, al problema de La Batalla Naval.
Por un lado, el Problema de Independent Set recibe un grafo como parámetro, con vértices, aristas y métodos (concretamente $grafo.obtener\_vertices()$ y $grafo.adyacentes(vertice)$), siendo N la cantidad de vertices de dicho grafo. Tambien se recibe un valor K, para el cual el problema implica devolver True si se comprueba que existe un Grafo Independent Set de al menos K vértices.
Por otro lado, el Problema de la Batalla Naval recibe como parámetro un tablero, un arreglo de barcos, un arreglo de demandas de filas y un arreglo de demandas de columnas.
Para realizar la reducción se propone:

- un tablero de dimensiones (N x N);
- un arreglo de N barcos, donde cada i, que representa cada longitud de barco, sea la cantidad de adyacentes al vertice i;
- un arreglo de N demandas de filas, donde cada j, que representa cada demanda de fila, sea la cantidad de adyacentes al vertice j;
- un arreglo de N demandas de columnas, donde cada j, que representa cada demanda de columna, sea un numero muy grande, de manera tal que no se lograse satisfacer.

Esto resulta en un tablero con la mayor cantidad de barcos colocados. Luego, se precisa de un validador (eficiente y que valide en tiempo polinomial) para saber cuantos barcos colocados hay. Cada barco es un vertice contenido en el Grafo de Independent Set. Como ultimo paso, se compara esa cantidad de barcos colocados con el valor K recibido por parámetro, devolviendose True si hay al menos K barcos (vertices) en el tablero (Grafo Independent Set).
De esta forma, queda demostrado que el Problema de la Batalla Naval es un problema NP-Completo.

5.3. El Problema de la Batalla Naval resuelto por Backtracking.

Para resolver el Problema de la Batalla Naval por Backtracking se tuvo en cuenta la versión de optimización del problema: dado un tablero de (n x m) casilleros, y una lista de k barcos (donde el barco i tiene bi de largo), una lista de las demandas de las n filas y una lista de las m demandas de las columnas, dar la asignación de posiciones de los barcos de tal forma que se reduzca al mínimo la cantidad de demanda incumplida.
A continuación se muestra el algoritmo en pseudocódigo, que detalla como se resolvió el problema:

```
def backtracking(tablero, barcos, d_filas, d_columnas):
    mejor_asignacion = [{}, float('-inf')]
    bt_recursivo(tablero, 0, ordenar_en_tuplas(barcos),
                 barcos_demasiados_largos(ordenar_en_tuplas(barcos), d_filas, d_columnas),
                 d_filas, d_columnas, d_filas.copy(), d_columnas.copy(), {}, mejor_asignacion,
                 sumar_longitudes(barcos))
    mejor_asignacion[0] = reconstruir_asignaciones(mejor_asignacion[0], len(barcos))
    return mejor_asignacion[0]
```

```
def bt_recursivo(pos_ocupadas, indice, barcos, barcos_a_omitir, d_filas, d_columnas,
d_filas_restantes, d_columnas_restantes, asignacion_actual, mejor_asignacion,
long_barcos_restante):
    if indice == len(barcos):
        demanda_cumplida_actual = calcular_demandas_cumplidas(asignacion_actual)
        if mejor_asignacion[1] < demanda_cumplida_actual:
            mejor_asignacion[0] = asignacion_actual.copy()
            mejor_asignacion[1] = demanda_cumplida_actual
        return

    if ya_no_llego(asignacion_actual, mejor_asignacion[1], long_barcos_restante):
        return

    t_barco = barcos[indice]

    if not barcos_a_omitir[indice]:
        for fil in range(len(d_filas)):
            for col in range(len(d_columnas)):
                if puedo_poner_barco(pos_ocupadas, t_barco, d_filas_restantes,
                d_columnas_restantes, fil, col, True, asignacion_actual):
                    poner_barco(pos_ocupadas, t_barco, d_filas_restantes, d_columnas_restantes,
                                asignacion_actual, fil, col, True)
                    bt_recursivo(pos_ocupadas, indice + 1, barcos, barcos_a_omitir, d_filas,
                                 d_columnas, d_filas_restantes, d_columnas_restantes,
                                 asignacion_actual, mejor_asignacion,
                                 long_barcos_restante - t_barco[1])
                    sacar_barco(pos_ocupadas, t_barco, d_filas_restantes, d_columnas_restantes,
                                asignacion_actual, fil, col, True)

                if puedo_poner_barco(pos_ocupadas, t_barco, d_filas_restantes,
                d_columnas_restantes, fil, col, False, asignacion_actual):
                    poner_barco(pos_ocupadas, t_barco, d_filas_restantes, d_columnas_restantes,
                                asignacion_actual, fil, col, False)
                    bt_recursivo(pos_ocupadas, indice + 1, barcos, barcos_a_omitir, d_filas,
                                 d_columnas, d_filas_restantes, d_columnas_restantes,
                                 asignacion_actual, mejor_asignacion,
```

```
                                long_barcos_restante - t_barco[1])
                    sacar_barco(pos_ocupadas, t_barco, d_filas_restantes, d_columnas_restantes,
                                asignacion_actual, fil, col, False)

    bt_recursivo(pos_ocupadas, indice + 1, barcos, barcos_a_omitir, d_filas, d_columnas,
                 d_filas_restantes, d_columnas_restantes, asignacion_actual, mejor_asignacion,
                 long_barcos_restante - t_barco[1])
```

- *ordenar_en_tuplas* ordena los barcos según su largo, y devuelve una lista de tuplas tal que cada tupla tiene la siguiente forma: (numero de barco, largo de barco);
- *calcular_demandas_cumplidas* calcula la cantidad de demandas cumplidas (tanto en la filas como en las columnas) por los barcos pasados por parámetro;
- *ya_no_llego* devuelve True si la asignación actual no puede superar la demanda satisfecha por la mejor asignación, por más que se inserten todos los barcos que todavía no se insertaron (Para esto se tuvo en cuenta que un barco de largo L satisface $2L$ de demanda, ya que el mismo satisface la demanda tanto de filas como de columnas);
- *puedo_poner_barco* evualua si se puede insertar un cierto barco que empieza desde una celda y está orientado de forma horizontal o vertical (Determinado por True (horizontal) o False (vertical)), y dadas ciertas celdas ya ocupadas;
- *poner_barco* inserta un cierto barco, empezando desde una celda y orientado de forma horizontal o vertical, y actualiza acordemente las celdas ya ocupadas y las demandas actuales;
- *sacar_barco* análogo al anterior, pero remueve el barco en vez de insertarlo

Teniendo ello se puede decir que, al ser este un algoritmo de Backtracking que se ejecuta a lo largo de ramas recursivas (cada ramificación corresponde a la decisión de poner un barco o no (o sea, 2 decisiones posibles)), la complejidad temporal del mismo es $O(2^n)$, siendo n la cantidad de barcos. A continuación se generaron sets de datos y mediciones de los mismos, para corroborar empíricamente esta complejidad.

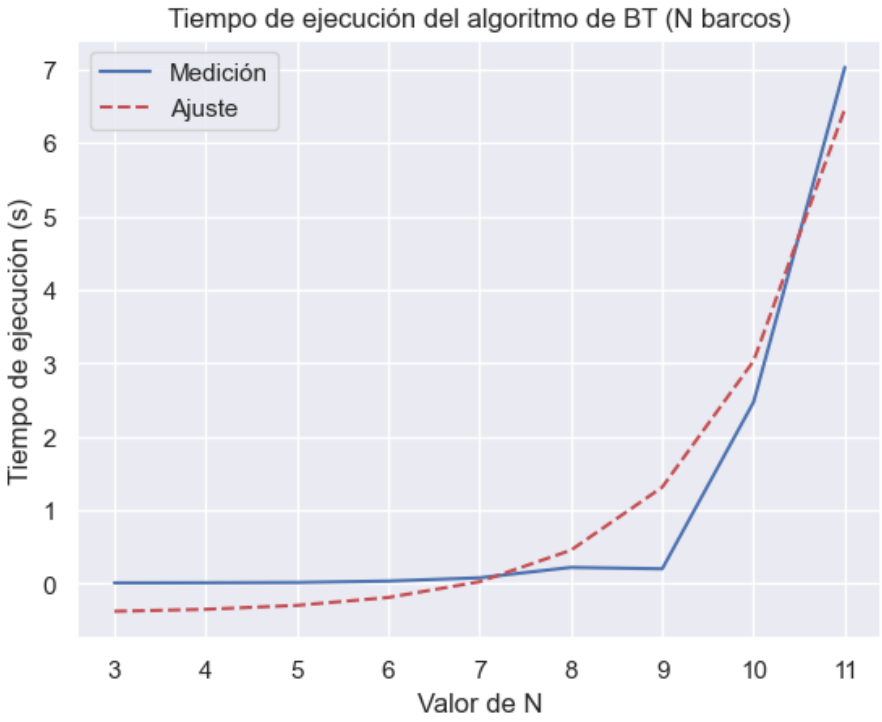


Figura 7: Tiempo de ejecución de Backtracking con Ajuste $O(2^n)$ (Usando n barcos, y una cantidad constante de filas y columnas).

Figura 8: Error Cuadrático Backtracking (Usando n barcos, y una cantidad constante de filas y columnas).

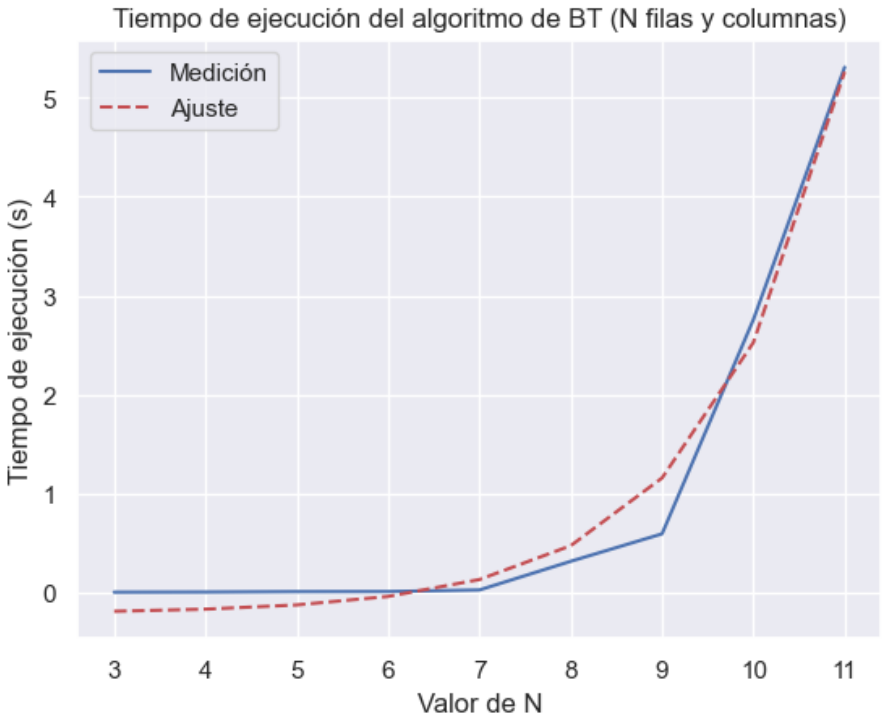


Figura 9: Tiempo de ejecución de Backtracking con Ajuste $O(2^n)$ (Usando n filas y columnas, y una cantidad constante de barcos).

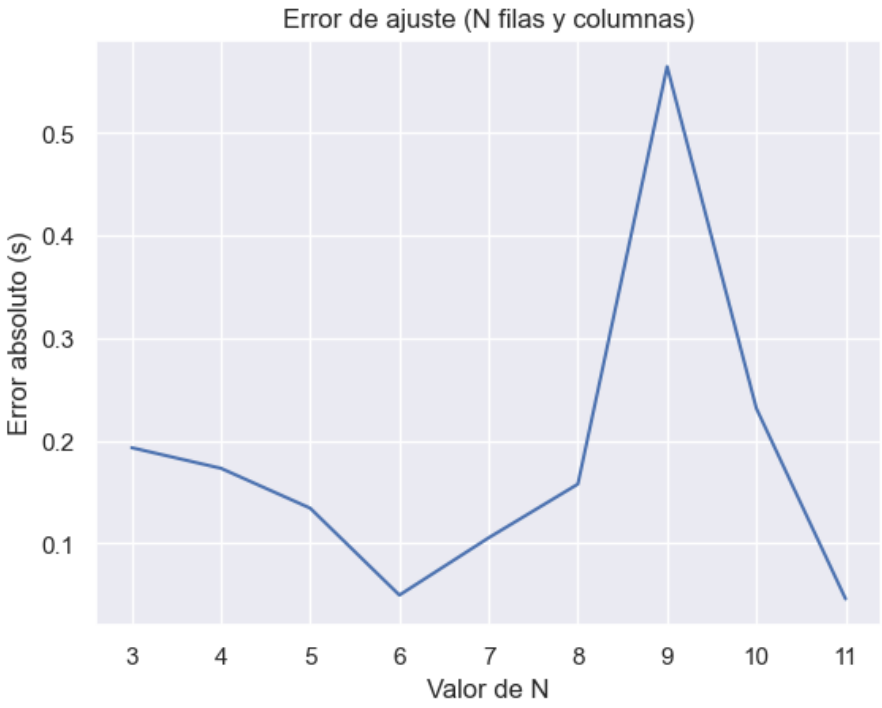


Figura 10: Error Cuadrático Backtracking (Usando n filas y columnas, y una cantidad constante de barcos).

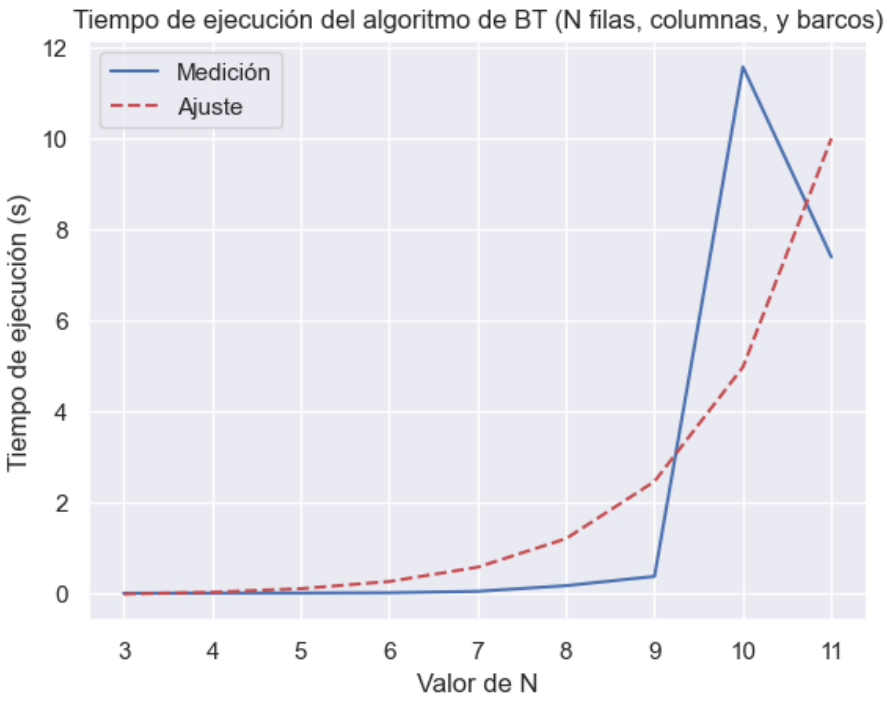


Figura 11: Tiempo de ejecución de Backtracking con Ajuste $O(2^n)$ (Usando n barcos, filas, y columnas).

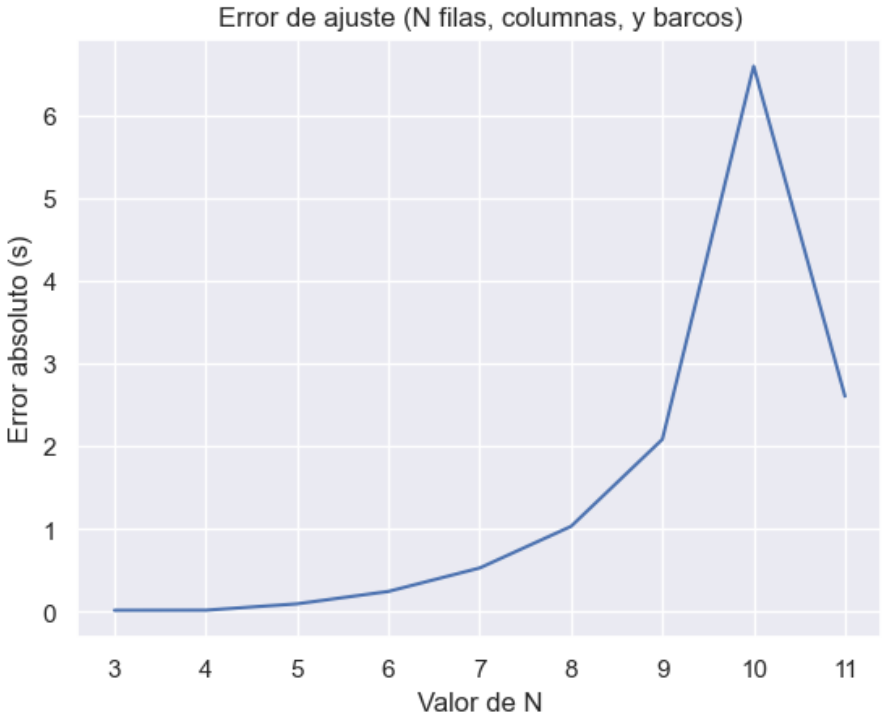


Figura 12: Error Cuadrático Backtracking (Usando n barcos, filas, y columnas).

5.4. El Problema de la Batalla Naval a través de un Modelo de Programación Lineal.

Se debe de escribir un modelo de Programación Lineal que resuelva el mismo problema anterior de forma óptima.

Se define las **variables** de interés para el modelo:

- $y_{b,i,j}$: variable binaria que representa si la posición (i, j) del tablero está ocupada por el barco b. Toma como valor 1 si la celda está ocupada, y 0 en caso contrario.

Se definen las **restricciones** del modelo:

- *restriccion de demandas por filas* : la suma de las posiciones ocupadas en cada fila i por todos los barcos no puede exceder la demanda especificada para esa fila, d_i:

$$\sum_{b,j} y_{b,i,j} \leq d_i, \forall i$$

- *restriccion de demandas por columnas* : la suma de las posiciones ocupadas en cada columna j por todos los barcos no puede exceder la demanda especificada para esa columna, d_j:

$$\sum_{b,i} y_{b,i,j} \leq d_j, \forall j$$

- *restriccion de adyacencia para barcos de longitud L* : si una celda (i, j) está ocupada por un barco de longitud l_b, entonces deben haber al menos $l_b - 1$ celdas ocupadas en la fila i y en la columna j, y a una distancia de como mucho $l_b - 1$ desde la celda (i, j):

$$\sum_{k=1}^{l_b-1} y_{b,i+k,j} + \sum_{k=1}^{l_b-1} y_{b,i-k,j} + \sum_{k=1}^{l_b-1} y_{b,i,j+k} + \sum_{k=1}^{l_b-1} y_{b,i,j-k} \geq (l_b - 1) \cdot y_{b,i,j}, \forall b, i, j$$

- *restriccion de longitud maxima de los barcos* : la cantidad de celdas ocupadas por un barco b no puede exceder su longitud especificada l_b:

$$\sum_{i,j} y_{b,i,j} \leq l_b, \forall b$$

- *restriccion de no adyacencia entre barcos* : una celda ocupada por un barco b' no puede estar adyacente (ni diagonalmente adyacente) a una celda ocupada por otro barco. Para esto se examina una grilla 3x3 alrededor de cada punto, con el mismo en el centro de la grilla:

$$\sum_{b|b\neq b',\lambda_1=-1}^{\lambda_1=1} \sum_{b|b\neq b',\lambda_2=-1}^{\lambda_2=1} y_{b,i+\lambda_1,j+\lambda_2} \leq 6 \cdot (1 - y_{b',i,j}), \forall b', i, j$$

  (La razón por la cual se puso un 6 ahí en vez de un 9 es que, dada cualquier grilla 3x3 que se seleccione en el tablero, es imposible que haya más de 6 celdas ocupadas por barcos sin que se rompa la regla de adyacencia. Además, poner un 9 en vez de un 6 hace que la restricción sea más exigente, lo cual hace que el algoritmo Simplex (basado en ramas recursivas similar a Backtracking) tenga ramas de ejecución mas cortas, y por lo tanto tarde menos tiempo)

La **función objetivo** busca maximizar la cantidad de espacios ocupados, respetando todas las restricciones anteriores:

$$\sum_{b,i,j} y_{b,i,j}, \forall b, i, j$$

Al ser este un algoritmo de Programación Lineal Entera el cual se resuelve con el método Simplex, la complejidad temporal del mismo es $O(2^n)$, siendo n un número proporcional a la cantidad de variables y restricciones creadas. A su vez, la cantidad de variables y restricciones creadas es proporcional a la cantidad de barcos, filas y columnas. A continuación se generaron sets de datos y mediciones de los mismos, para corroborar empíricamente esta complejidad

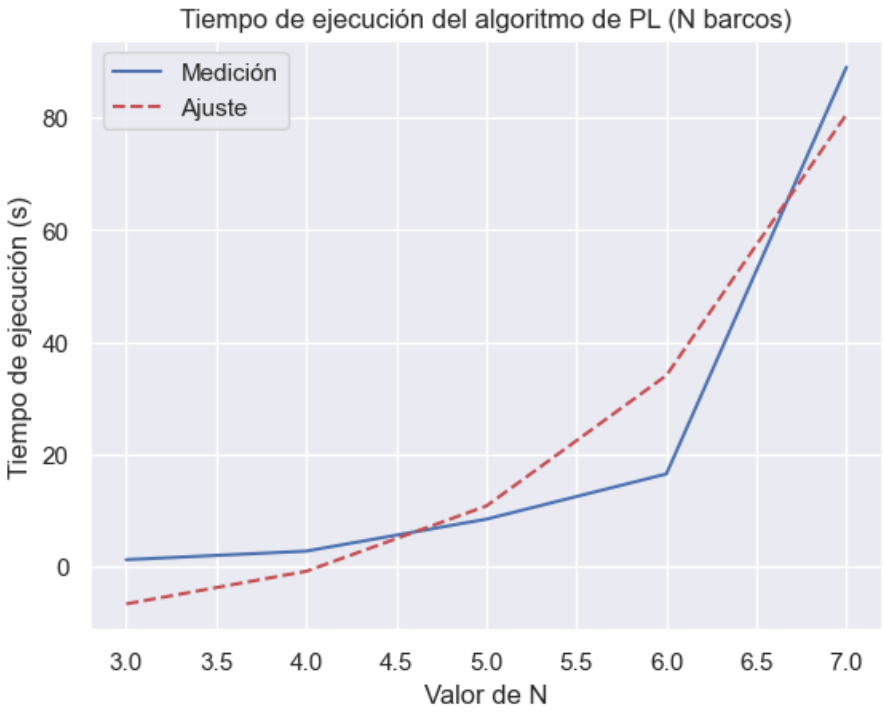


Figura 13: Tiempo de ejecución de PLE con Ajuste $O(2^n)$ (Usando n barcos, y una cantidad constante de filas y columnas).

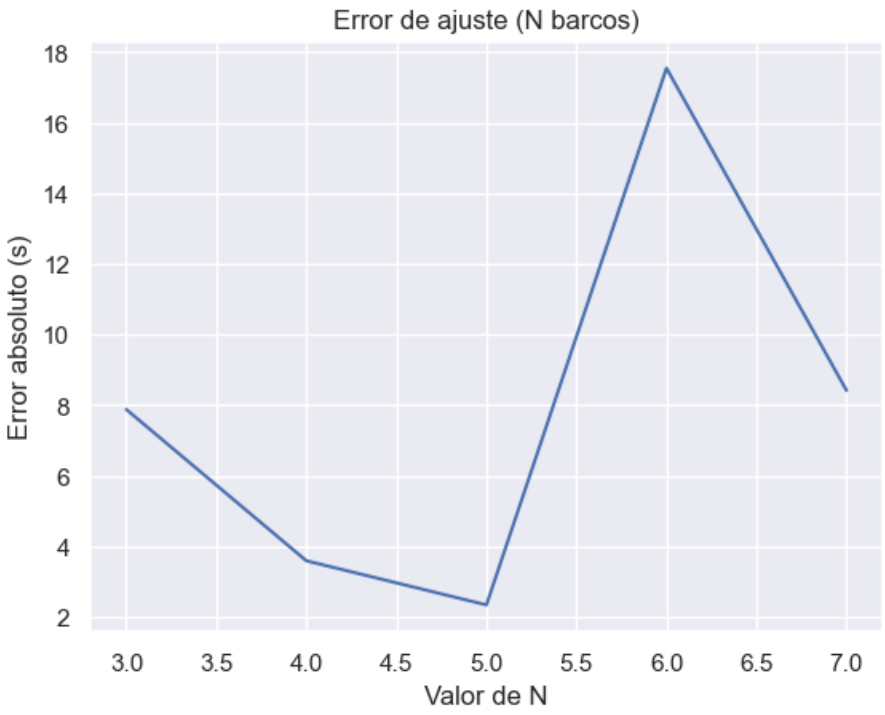


Figura 14: Error Cuadrático PLE (Usando n barcos, y una cantidad constante de filas y columnas).

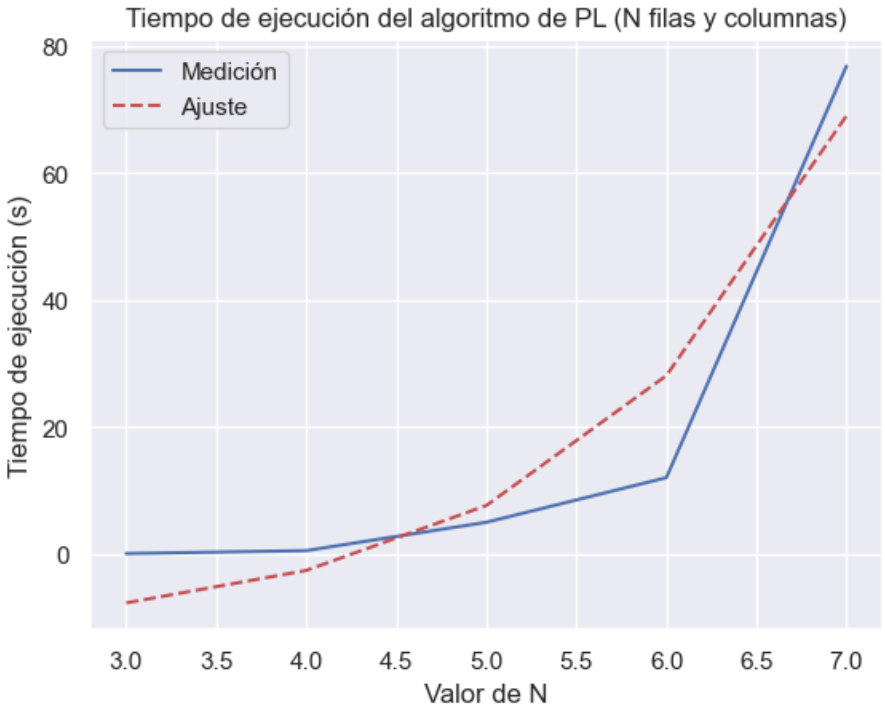


Figura 15: Tiempo de ejecución de PLE con Ajuste $O(2^n)$ (Usando n filas y columnas, y una cantidad constante de barcos).

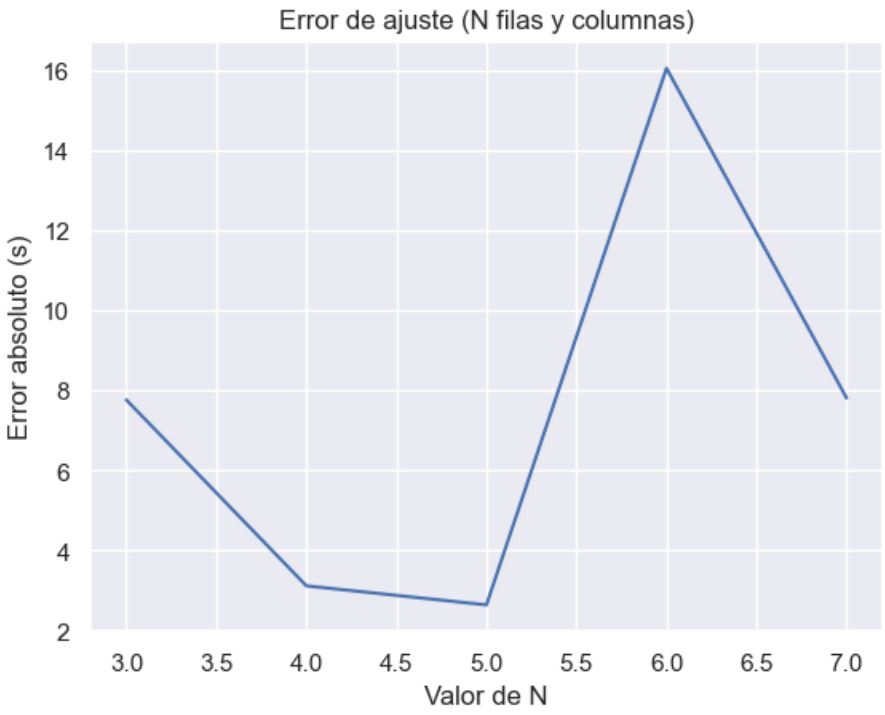


Figura 16: Error Cuadrático PLE (Usando n filas y columnas, y una cantidad constante de barcos).

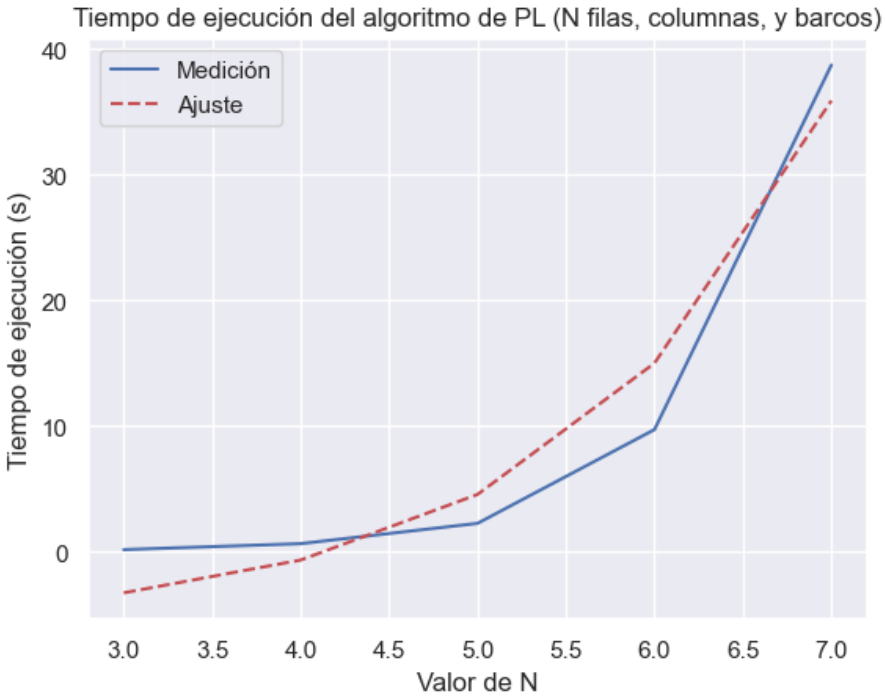


Figura 17: Tiempo de ejecución de PLE con Ajuste $O(2^n)$ (Usando n barcos, filas, y columnas).

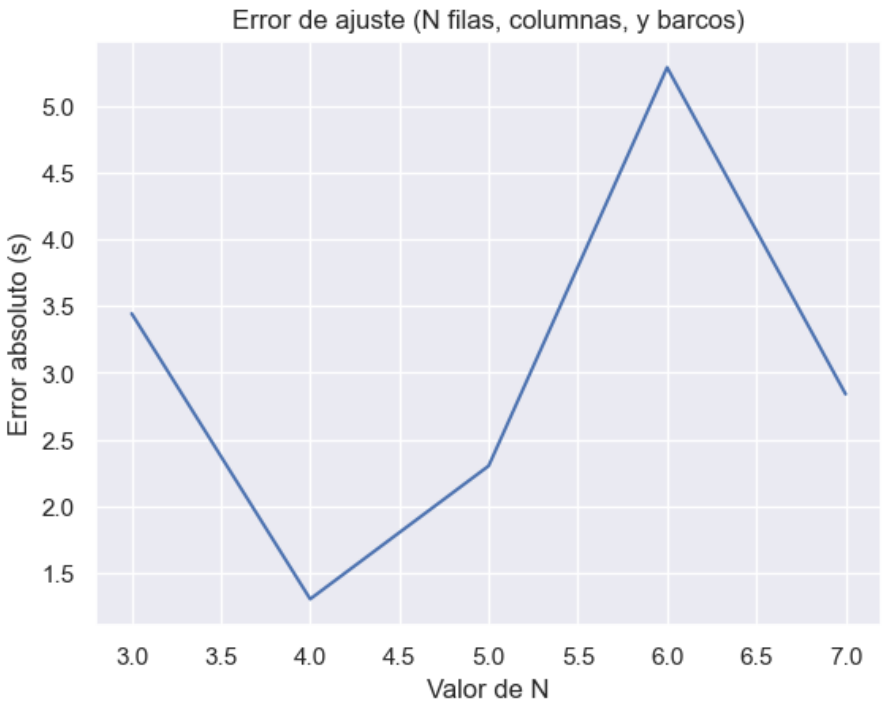


Figura 18: Error Cuadrático PLE (Usando n barcos, filas, y columnas).

5.5. Algoritmo de Aproximación: John Jellicoe.

John Jellicoe (almirante de la Royal Navy durante la batalla de Jutlandia) nos propone el siguiente algoritmo de Aproximación:

- ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido;
- si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente;
- volver a aplicar hasta que no queden más barcos o no haya más demandas por cumplir.

En este caso, el objetivo será maximizar la cantidad de asignación de posiciones de los barcos en el tablero, de forma que se obtenga la mayor cantidad de demanda satisfecha posible. Este enfoque se representa un algoritmo de Aproximación que, para su implementación, propone una solución Greedy. A través de diferentes funciones, que ejecutan ciclos iterativos, es que se buscara resolver el Problema de la Batalla Naval.

```
1 def johnjellicoe(tablero, barcos, d_filas, d_columnas):
2     tablero_resultado = copy(tablero)
3     d_filas_restantes = copy(d_filas)
4     d_columnas_restantes = copy(d_columnas)
5     asignaciones = colocar_barcos(tablero_resultado, sorted(barcos, reverse=True),
6     d_filas_restantes, d_columnas_restantes)
7     return asignaciones
```

```python
def colocar_barcos(tablero, barcos, d_filas_restantes, d_columnas_restantes):
    asignaciones = {i: None for i in range(len(barcos))}

    for i, barco in enumerate(barcos):
        colocado = False
        fila = seleccionar_fila(d_filas_restantes)
        columna = seleccionar_columna(d_columnas_restantes)

        if d_filas_restantes[fila] >= d_columnas_restantes[columna]:
            for col in range(len(tablero[0])):
                if puedo_colocar_el_barco(tablero, d_filas_restantes,
                        d_columnas_restantes, fila, col, barco, True):
                    colocar_barco(tablero, d_filas_restantes, d_columnas_restantes,
                        fila, col, barco, True)
                    asignaciones[i] = [(fila, col), (fila, col + barco - 1)]
                    colocado = True
                    break
        else:
            for fila in range(len(tablero)):
                if puedo_colocar_el_barco(tablero, d_filas_restantes,
                        d_columnas_restantes, fila, columna, barco, False):
                    colocar_barco(tablero, d_filas_restantes, d_columnas_restantes,
                        fila, columna, barco, False)
                    asignaciones[i] = [(fila, columna), (fila + barco - 1, columna)]
                    colocado = True
                    break
        if not colocado:
            continue

    return asignaciones
```

La complejidad del algoritmo John Jellicoe se puede analizar de la siguiente manera:

- ordenar los k barcos por longitud de mayor a menor tiene una complejidad $O(k\ log\ k)$;
- el algoritmo intenta colocar k barcos, cada uno con una complejidad de $O(n+m)$. Lo que resulta en $O((n+m)\cdot k)$;
- en cada fila o columna, el algoritmo evalúa si puede insertar barcos de largo L. Cada evaluación examina las demandas de filas y columnas (para así no superarlas), y examina adyacencia con otros barcos. Estas dos acciones se realizan en un tiempo constante proporcional a L. Esto resulta en complejidad $O(1)$;
- si se desea imprimir el tablero esto implicaría recorrer todas las posiciones, lo cual tiene una complejidad $O(n\cdot m)$;

Dado que k es el número de barcos, y n, m las dimensiones del tablero, la complejidad total del algoritmo es: $O(k\cdot log(k)+(n+m)\cdot k)$, la cual se puede resumir a $O((n+m)\cdot k)$. En caso de que se desee imprimir el tablero, la complejidad del algoritmo aumentaría a $O(n\cdot m)$.

Por último, se debe analizar cuán buena aproximación es este algoritmo respecto a la solución óptima. Para esto se define:

- sea I cualquier instancia del problema de La Batalla Naval;
- sea $A(I)$ la demanda cumplida por el algoritmo aproximado para I;

- sea $z(I)$ la demanda cumplida por la solución óptima para I.

La razón de aproximación $r(A)$ se define como:

$$\frac{A(I)}{z(I)} \geq r(A)$$

Mediante mediciones empíricas, las cuales compararon varias instancias de problemas y sus resoluciones con el algoritmo exacto y el algoritmo aproximado, se pudo concluir que un valor aproximado de $r(A)$ es el siguiente:

$$r(A) \approx 0{,}279$$

5.6. Algoritmo de Aproximación: Barcos Grandes Primero.

Se implementa el algoritmo de aproximación Barcos Grandes Primero para comparar su performance versus el algoritmo John Jellicoe. De entrada, se toma esta elección porque se cree que este algoritmo Greedy podría tener las siguientes ventajas:

- facilitar la colocación de los barcos más restrictivos primero con el objetivo de que, en caso de los mismos no puedan ser colocados, se cuente con la opción de colocar barcos más pequeños los cuales tienen mayor posibilidad de poder ser colocados;
- maximizar lo antes posible la demada cubierta;
- eliminar el riesgo de que colocar un barco pequeño imposibilite la colocación de barcos más grandes que podrían satisfacer más demanda;

El algoritmo propuesto sigue la regla de colocar primero los barcos grandes en la primera posición válida, cumpliendo con las restricciones del tablero y las demandas restantes. Esto lo hace priorizando las decisiones locales inmediatas y esperando que dichas decisiones conduzcan a una óptima solución global.

```
def barcos_grandes_primero(tablero, barcos, d_filas, d_columnas):
    tablero_resultado = copy(tablero)
    d_filas_restantes = copy(d_filas)
    d_columnas_restantes = copy(d_columnas)
    asignaciones = colocar_barcos(tablero_resultado,
                   sorted(barcos, reverse=True), # Ordeno barcos de mayor a menor longitud
                   d_filas_restantes, d_columnas_restantes)
    return asignaciones
```

```
def colocar_barcos(tablero, barcos, d_filas_restantes, d_columnas_restantes):
    asignaciones = {i: None for i in range(len(barcos))}

    for i, barco in enumerate(barcos):
        colocado = False
        for fila in range(len(tablero)):
            for columna in range(len(tablero[0])):
                if puedo_colocar_el_barco(tablero, d_filas_restantes, d_columnas_restantes,
                        fila, columna, barco, True):
                    colocar_barco(tablero, d_filas_restantes, d_columnas_restantes, fila,
                        columna, barco, True)
                    asignaciones[i] = [(fila, columna), (fila, columna + barco - 1)]
                    colocado = True
                    break
```

```
                if puedo_colocar_el_barco(tablero, d_filas_restantes, d_columnas_restantes,
                        fila, columna, barco, False):
                    colocar_barco(tablero, d_filas_restantes, d_columnas_restantes, fila,
                        columna, barco, False)
                    asignaciones[i] = [(fila, columna), (fila + barco - 1, columna)]
                    colocado = True
                    break
            if colocado:
                break

    return asignaciones
```

La complejidad del algoritmo Barcos Grandes Primero se puede analizar de la siguiente manera:

- ordenar los k barcos por longitud de mayor a menor tiene una complejidad $O(k\ log\ k)$;
- recorrer todas las celdas del tablero (n filas y m columnas), una vez por cada uno de los k barcos, tiene una complejidad $O(k \cdot n \cdot m)$.
- en cada celda, el algoritmo evalúa si puede insertar barcos de largo L. Cada evaluación examina las demandas de filas y columnas (para así no superarlas), y examina adyacencia con otros barcos. Estas dos acciones se realizan en un tiempo constante proporcional a L. Esto resulta en complejidad $O(1)$;

Dado que k es el número de barcos, y n, m las dimensiones del tablero, la complejidad total del algoritmo es $O(k \cdot log(k) + k \cdot n \cdot m)$, la cual se puede resumir a $O(k \cdot n \cdot m)$.

Siguiendo el mismo proceso que el algoritmo de aproximación John Jellicoe, para el caso de esta aproximación A se obtuvo el siguiente valor aproximado de $r(A)$:

$$r(A) \approx 0{,}7209$$

5.7. Conclusiones.

Se ha demostrado la complejidad y diversidad de enfoques para resolver el Problema de la Batalla Naval, en especial la variante de problema de optimización. Por un lado, se vizualizaron algoritmos que resuelven el problema llegando a la mejor solución posible, tales como lo son Backtracking y Programación Lineal, observando que BT resolvió el problema más rápido que PL. Por el otro, se analizaron y propusieron algoritmos que brindan una aproximación, se calculó como se comparan los mismos con los algoritmos óptimos, y se logró la conclusión que la aproximación Barcos Grandes Primero es mejor que el algoritmo de John Jellicoe.

6. Anexo de correcciones.

1.
   - Demostración que el algoritmo Greedy planteado es óptimo. Caso base:
     - $i = f \longrightarrow GR(i, i) = V[i]$ que es óptimo porque es la unica moneda del arreglo, y por lo tanto el máximo valor;
     - $i = f - 1 \longrightarrow GR(i, i + 1) = max(V[i], V[i + 1])$ que es óptimo porque se toma la moneda de mayor valor entre las dos posibles.
   - Caso inductivo: Si $GR(i + 1, f - 1)$, $GR(i + 2, f)$ y $GR(i, f - 2)$ son óptimos (terminan en victoria para Sophia o en empate), entonces $GR(i, f)$ es óptimo

- Demostración: la ecuación recursiva es la siguiente:

$$GR(i,f) = max \left\{ \begin{array}{l} V[i] + \left\{ \begin{array}{ll} GR(i+1,f-1) & \text{si V[i+1] >= V[f],} \\ GR(i+2,f) & \text{de lo contrario} \end{array} \right\} \\ V[f] + \left\{ \begin{array}{ll} GR(i+1,f-1) & \text{si V[i] <= V[f-1],} \\ GR(i,f-2) & \text{de lo contrario} \end{array} \right\} \end{array} \right\} \quad (1)$$

Como $GR(i+1,f-1)$, $GR(i+2,f)$ y $GR(i,f-2)$ se suponen como óptimos y la función recursiva es un max, esto significa que a uno de esos óptimos se le va a sumar la moneda más grande posible entre $V[i]$ y $V[f]$. Por lo tanto, $GR(i,f)$ va a tener el mayor valor posible que va a resultar en victoria para Sophia o en empate, por ende $GR(i,f)$ es óptimo.

Como el caso base y el inductivo son válidos, el algoritmo es óptimo

2. Ecuación de recurrencia de la parte 2 del trabajo práctico legible:

$$OPT(i,f) = max \left\{ \begin{array}{l} V[i] + \left\{ \begin{array}{ll} OPT(i+1,f-1) & \text{si V[i+1] <V[f],} \\ OPT(i+2,f) & \text{de lo contrario} \end{array} \right\} \\ V[f] + \left\{ \begin{array}{ll} OPT(i+1,f-1) & \text{si V[i] >V[f-1],} \\ OPT(i,f-2) & \text{de lo contrario} \end{array} \right\} \end{array} \right\} \quad (2)$$

3. Mediciones por separado de los tiempos de ejecución del algoritmo de Programación Dinámica y su reconstrucción.

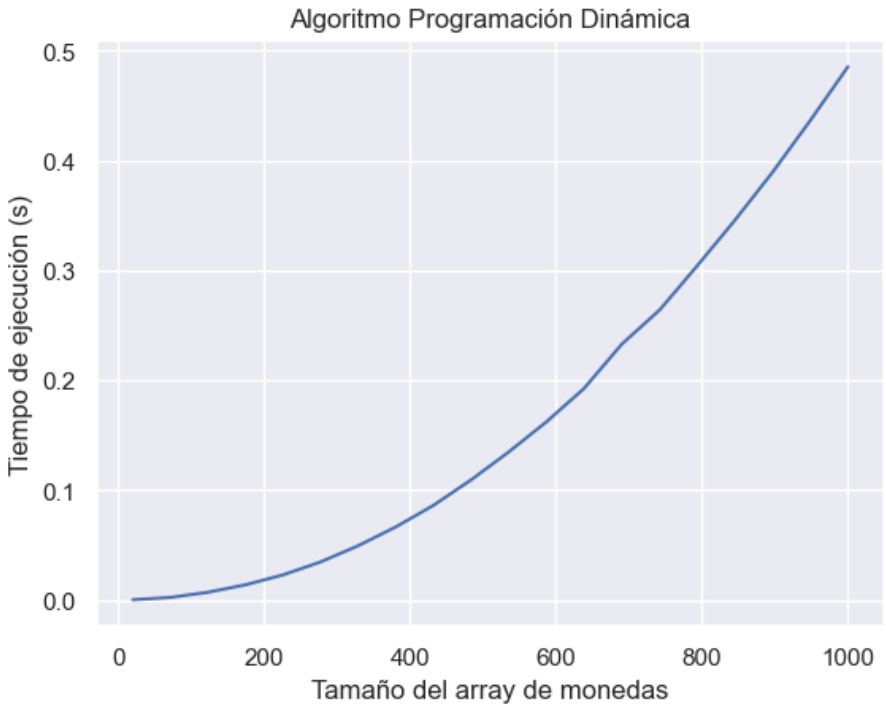


Figura 19: Tiempo de ejecución de PD (construcción de tabla de óptimos)

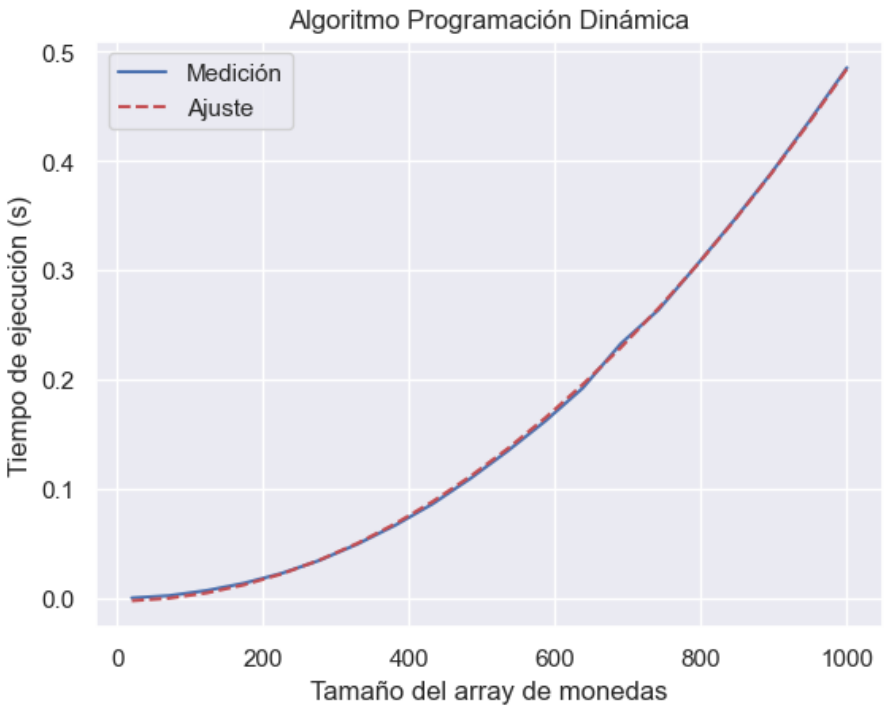


Figura 20: Tiempo de ejecución de PD (construcción de tabla de óptimos) con Ajuste $O(n^2)$

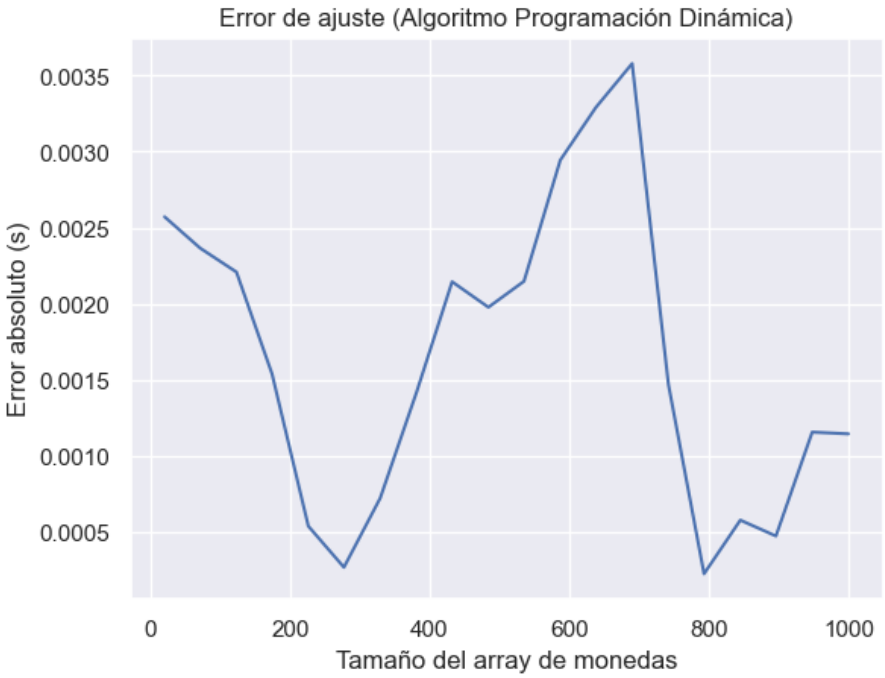


Figura 21: Error Cuadrático del Algoritmo PD (construcción de tabla de óptimos)

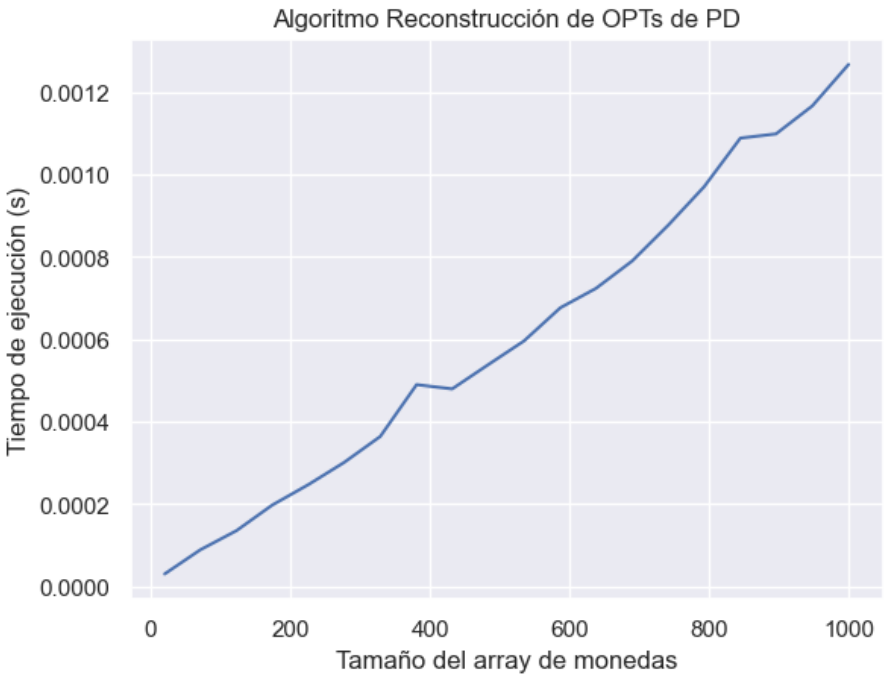


Figura 22: Tiempo de ejecución de Reconstrucción

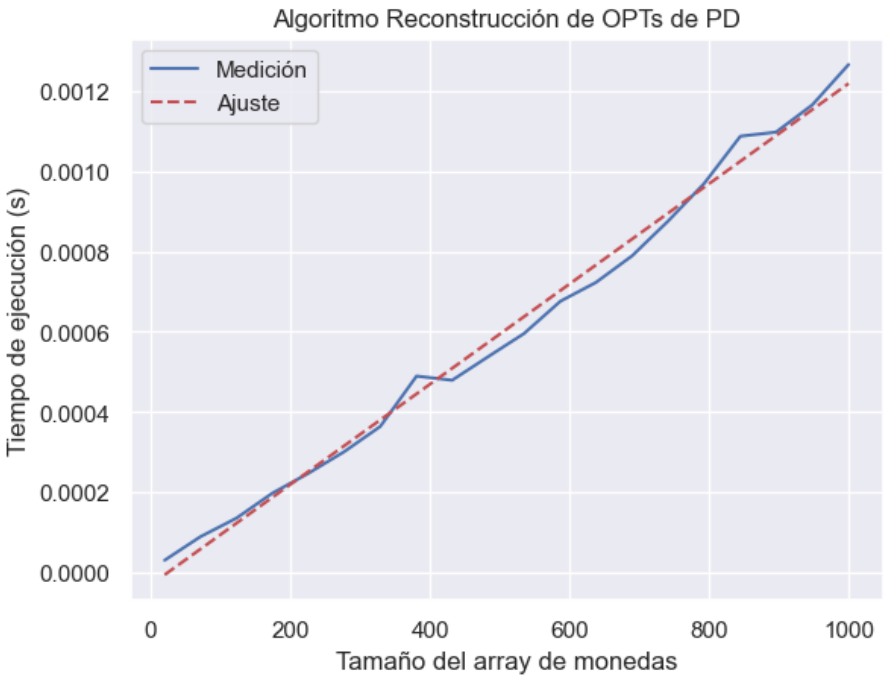


Figura 23: Tiempo de ejecución de Reconstrucción con Ajuste $O(n)$

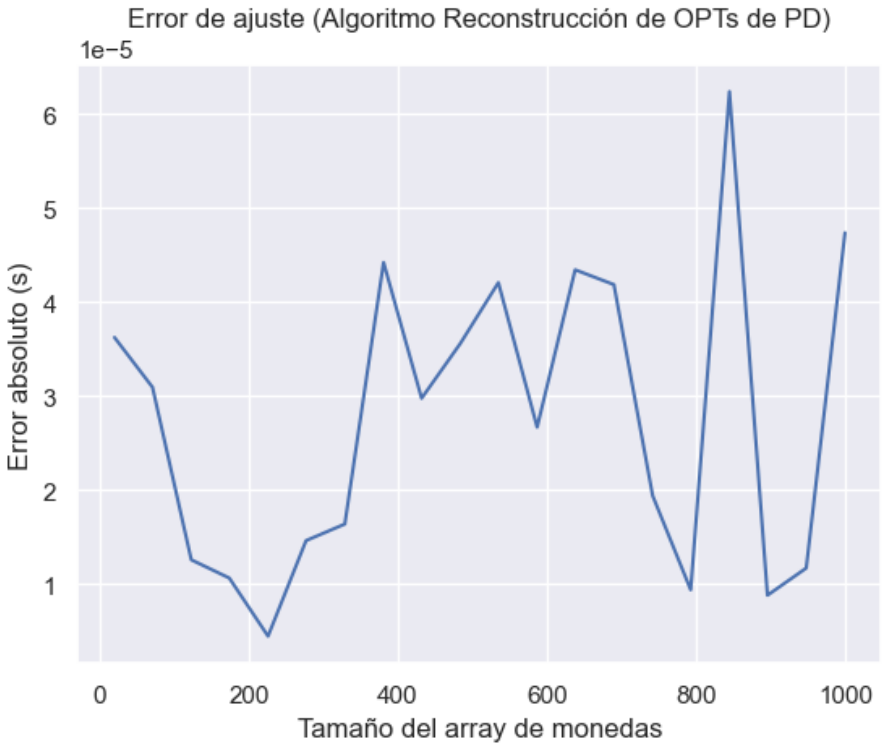


Figura 24: Error Cuadrático de Reconstrucción

4. Validador para demostrar que el problema está en NP. Dicho validador ha de validar en tiempo poliniomial y comprobar que las demandas no se encuentren insatisfechas y/o sobre-satisfechas:

```
def validar(barcos_disponibles, demanda_filas, demanda_columnas, asignaciones):
    if len(barcos_disponibles) != len(asignaciones):
        return False

    tablero = [[False for _ in range(len(demanda_columnas))]
               for _ in range(len(demanda_filas))]
    filas = demanda_filas.copy()
    columnas = demanda_columnas.copy()

    # Colocar barcos para comprobar colisiones y adyacencia
    for key in asignaciones.keys():
        coords = asignaciones[key]
        if coords is not None:
            if not se_puede_poner_barco(coords[0], coords[1], tablero):
                return False

    # Contar demandas satisfechas
    for i in range(len(tablero)):
        for j in range(len(tablero[i])):
            if tablero[i][j]:
                filas[i] -= 1
                columnas[j] -= 1

```

```
24                  # Buscar demandas sobresatisfechas y/o demandas insatisfechas
25                  for num in filas:
26                      if num < 0 or num > 0:
27                          return False
28                  for num in columnas:
29                      if num < 0 or num > 0:
30                          return False
31                  return True
```

- *se_puede_poner_barco* evalúa si se puede colocar el barco en el tablero (horizontal o verticalmente), en base a el tamaño del propio barco y sus adyacencias, ignorando las demandas;
- La complejidad de este validador se puede analizar de la siguiente manera:
  - Generar el tablero de n filas y m columnas para luego chequear colisiones, adyacencias y demandas tiene una complejidad $O(n * m)$
  - Comprobar colisiones y adyacencia de los k barcos (que tienen en el peor de los casos largo n o m) tiene una complejidad $O(k * n)$ o $O(k * m)$ (se elige la peor de las dos)
  - Contar las demandas satisfechas de las n filas y m columnas tiene una complejidad $O(n * m)$
  - Chequear si las demandas contadas están sobresatisfechas o insatisfechas tiene una complejidad $O(n + m)$

  En resumen, se puede decir que el validador tiene una complejidad aproximadamente igual a $O(n * m)$, siendo n la cantidad de filas y m la cantidad de columnas

5. Para demostrar que el problema de decisión de Batalla Naval es NP-C, se debe reducir la siguiente versión del Problema de decisión de Bin-Packing: se tiene un set finito S de n elementos, cada uno con un volumen $T : S \longrightarrow Z^+$ (siendo Z^+ el conjunto de los enteros positivos), y K contenedores de capacidad C, y se quiere saber: ¿se pueden colocar todos los elementos de S en los K contenedores tal que la sumatoria de todos los volúmenes de los elementos dentro de cada contenedor sea igual a C?

   Para esta reducción, se considera Bin Packing con parámetros unarios: Los volúmenes, K y C son números escritos en unario. Esto evitará que la reducción sea pseudo-polinomial y por lo tanto inválida.

   Reducción de Bin-Packing al Problema de la Batalla Naval:

   - Para cada elemento i de Bin-Packing, se crea un barco de largo $T(i)$ en el arreglo de barcos ($O(n)$)
   - Se instancia el siguiente tablero ($O(n * K)$):
     - La cantidad de filas es igual a la sumatoria de todos los elementos en S mas $n - 1$;
     - La demanda de las filas es la siguiente: si $S = \{i_0, i_1, ..., i_{n-1}\}$, las primeras $T(i_0)$ filas tienen todas demanda 1, luego la siguiente fila a esas tiene demanda 0, luego las siguientes $T(i_1)$ filas tienen todas demanda 1, luego la siguiente tiene demanda 0, y así sucesivamente hasta rellenar las demandas de todas las filas;
       - EJ: $T(S) = \{5, 2, 3\} \Longrightarrow DemandasDeFilas = [1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]$
       - La razón de esta configuración de demandas de filas es que queremos que se puedan colocar todos los barcos sin que estos tengan problemas de adyacencia
     - La cantidad de columnas es $2K - 1$;
     - Las demandas de las columnas son C para las columnas impares y 0 para las columnas pares.

- Se llama 1 sola vez al resolvedor de Problema de Batalla Naval ($O(1)$)
- Si el resolvedor devuelve True, entonces la solución a Bin Packing es True. Caso contrario, la solución es False ($O(1)$)

Demostración de "La solución al problema de Bin Packing es True $\iff$ La solución al problema de la Batalla Naval es True":

- La solución de Bin Packing es True $\implies$ La solución de Batalla Naval es True
  - Si Bin Packing es True, eso significa que todos los elementos del set S se pudieron repartir en K contenedores, de tal forma que la sumatoria de volumenes de los elementos en cada contenedor es igual a C.
  - A priori, los barcos que coloquemos nunca van a colisionar debido a la existencia de las filas y columnas de demanda 0
  - Los barcos correspondientes a los elementos de cada uno de los K contenedores van a colocarse en cada una de las K columnas con demanda C, de tal forma que cada una de ellas solo tenga barcos correspondientes a un mismo contenedor. Ya que los elementos de cada contenedor suman C, eso significa que la demanda C de estas columnas se va a poder satisfacer.
  - Por otro lado, las filas de demanda 1 van a ser satisfechas ya que como se tuvo que utilizar todos los elementos para poder llenar los K contenedores de capacidad C, esto significa que inevitablemente se tienen que utilizar todos los barcos (que representan a los volúmenes de los elementos) para poder satisfacer las demandas de todas las columnas, lo cual significa que las demandas de todas las filas también se podrán satisfacer (recordar que utilizar todos los barcos es la única forma de satisfacer estas demandas)
  - Como todas las demandas son satisfechas, la solución al Problema de Batalla Naval es True
- La solución de Bin Packing es True $\impliedby$ La solución de Batalla Naval es True
  - Si Batalla Naval es True, eso significa que:
    - Se satisfacieron las demandas de todas las filas, por ende se tuvieron que usar todos los barcos
    - Se satisfacieron las demandas de todas las columnas, por ende todos los barcos se pudieron distribuir en las K columnas de demanda C
  - Como todos los barcos representan los volúmenes de los elementos de S, el anterior item significa que se pudo organizar todos estos elementos en K grupos de volumen C, lo cual significa que la solución a Bin Packing es True

Bin Packing es NP Completo

Se puede demostrar que Bin Packing unario es NP Completo reduciendo polinomialmente el problema de desición de 2-partition binario: Se tiene un multiset finito S' de m números enteros positivos, y se quiere saber ¿Se puede particionar S' en dos conjuntos disjuntos, tal que la sumatoria de los números de ambos conjuntos sea la misma?

Reducción de 2-Partition a Bin Packing:

- Definimos $K = 2$ (dos contenedores) ($O(1)$)
- Definimos C igual a la sumatoria de todos los elementos en S', dividido por 2 ($O(m)$)
- Convertimos todos los m números binarios de S' a volúmenes unarios para el conjunto S de Bin Packing ($O(m)$)

Por otro lado, la demostración del siguiente postulado:

2-Partition binario es True $\iff$ Bin Packing unario es True

Es trivial, ya que 2-Partition es un caso particular de Bin Packing, donde $K = 2$ y C es igual a la sumatoria de todos los elementos en S' dividido por 2, por lo que un Bin packing con estos parámetros se comportará exactamente igual a 2-Partition. Luego, la conversión de binario a unario y viceversa también es trivial.

6. Comparaciones de tiempos de ejecución entre Programación Lineal y Backtracking.

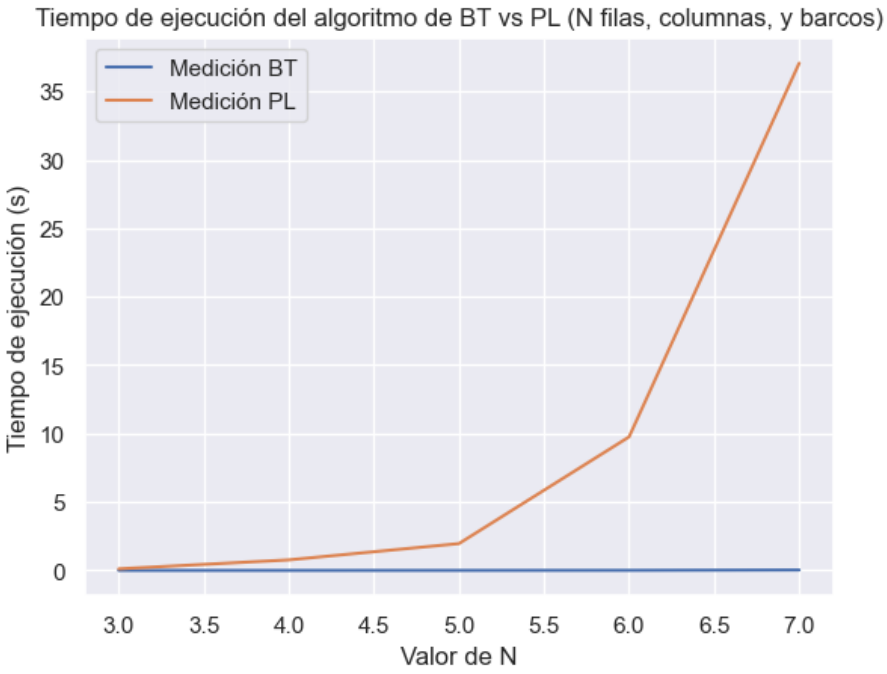
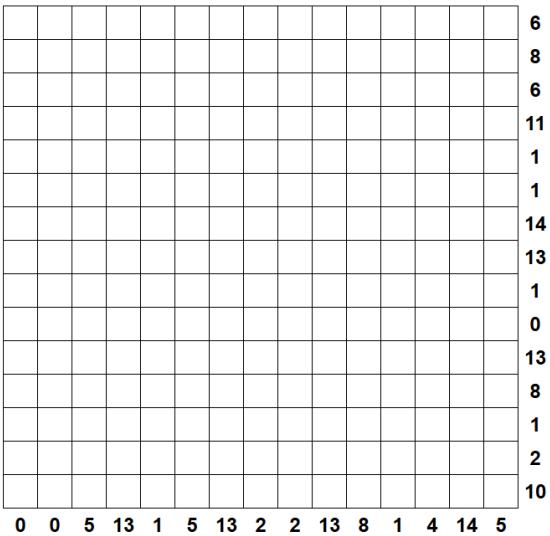


Figura 25: Tiempo de ejecución de Backtracking vs. Tiempo de ejecución de PLE (Usando n barcos, filas y columnas).

7. La instancia que produjo la menor cota de aproximación posible, tanto para el algoritmo John Jellicoe como para el algoritmo greedy Barcos Grandes Primero, es la siguiente:




9
4
14
8
10
14
12
6
12
14
3
9
6
1
5