.UBAfiuba
FACULTAD DE INGENIERÍA

TRABAJO PRÁCTICO 1

TEORÍA DE ALGORITMOS

(75.29 / 95.06 / TB024)

Cuatrimestre: 2º Cuatrimestre 2024

Fecha de entrega: 29 / 09 / 2024

Grupo: 5 (She Don't Give Grafo)

Curso: 03 - Echevarría

Integrantes:

Pablo Choconi	**106388**
Valentín Savarese	**107640**
Gian Luca Spagnolo	**108072**
Brian Céspedes	**108219**
Néstor Palavecino	**108244**

Índice

1. PROBLEMA 1

1.1. Enunciado y Supuestos

Se tiene una bolsa con **n** monedas de idéntica denominación, de las cuales exactamente una es falsa y se sabe que es más liviana que el resto.

- Diseñar e implementar un algoritmo de División y Conquista que identifique la moneda falsa, sin recurrir a ordenar las monedas por su peso
- NOTA: utilizar tamaños de datos con n = {10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000}.

También, tenemos en cuenta los siguientes supuestos:

- El número n debe ser al menos >= 2, o sea, el caso base serían 2 monedas. Esto es debido a que no tiene sentido tener una sola moneda y catalogarla como falsa o verdadera, sin algún criterio de comparación, frente a por lo menos otra moneda.

1.2. Diseño de la Solución

Para este ejercicio, hemos diseñado un algoritmo de **División y Conquista**, con funcionamiento similar a *Mergesort*. Nuestro planteo se basa en la división del problema general (en este caso, una bolsa de monedas) en subproblemas más pequeños cada vez, como si se tratase de un árbol, hasta finalmente llegar a los "nodos hoja" siguiendo esta analogía.

El objetivo de este funcionamiento es poder comparar individualmente cada par de elementos con el objetivo de encontrar finalmente una moneda de peso menor al resto.

1.3. Implementación del Algoritmo

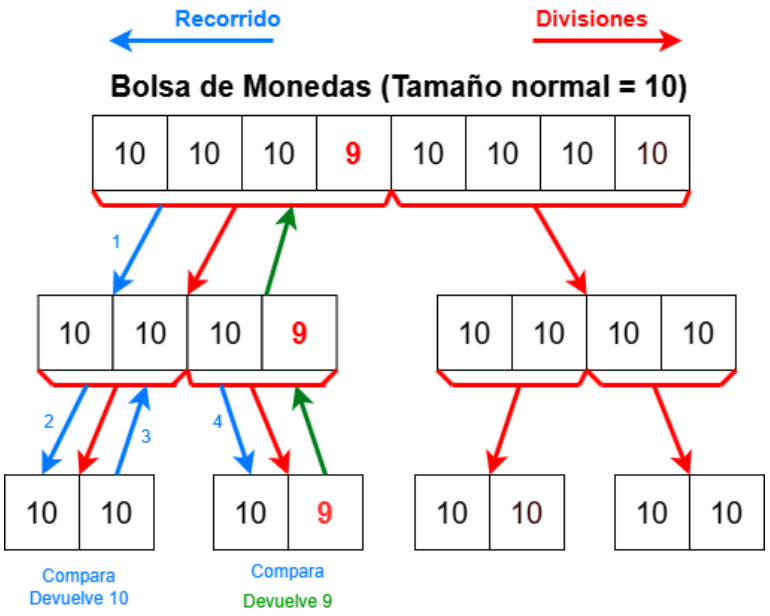
```python
def detectar_monedas_falsa_dyc(monedas, start_index, end_index):
    # Caso base
    if end_index - start_index <= 1:
        if monedas[start_index] < monedas[end_index]:
            return monedas[start_index]
        else:
            return monedas[end_index]

    # Caso recursivo
    mid_index = start_index + (end_index - start_index) // 2
    moneda_izq = detectar_monedas_falsa_dyc(monedas, start_index, mid_index)
    if moneda_izq < monedas[mid_index + 1]:
        return moneda_izq
    moneda_der = detectar_monedas_falsa_dyc(monedas, mid_index + 1, end_index)
    return moneda_der
```

En nuestra implementación, consideramos a la bolsa de monedas como una lista del peso de cada moneda, donde se busca dividir en mitades esta lista hasta llegar al límite de la recursión y, al tener dos monedas a disposición, compararlas para encontrar cual de estas es la falsa (en caso de que alguna lo sea).

Inicialmente, habíamos diseñado un algoritmo de similar funcionamiento pero que no cumplía en su totalidad con la regla de algoritmo de División y Conquista. Este aprovechaba que, sabiendo la cantidad total de monedas y el peso de una moneda normal, ponderar por mitades del conjunto de monedas encontrando una mitad con un peso menor al esperado, hasta ir reduciendo el problema y encontrarlo. Sin embargo, a pesar de que no cumple con la principal condición del enunciado, este algoritmo resultó ser sumamente veloz.

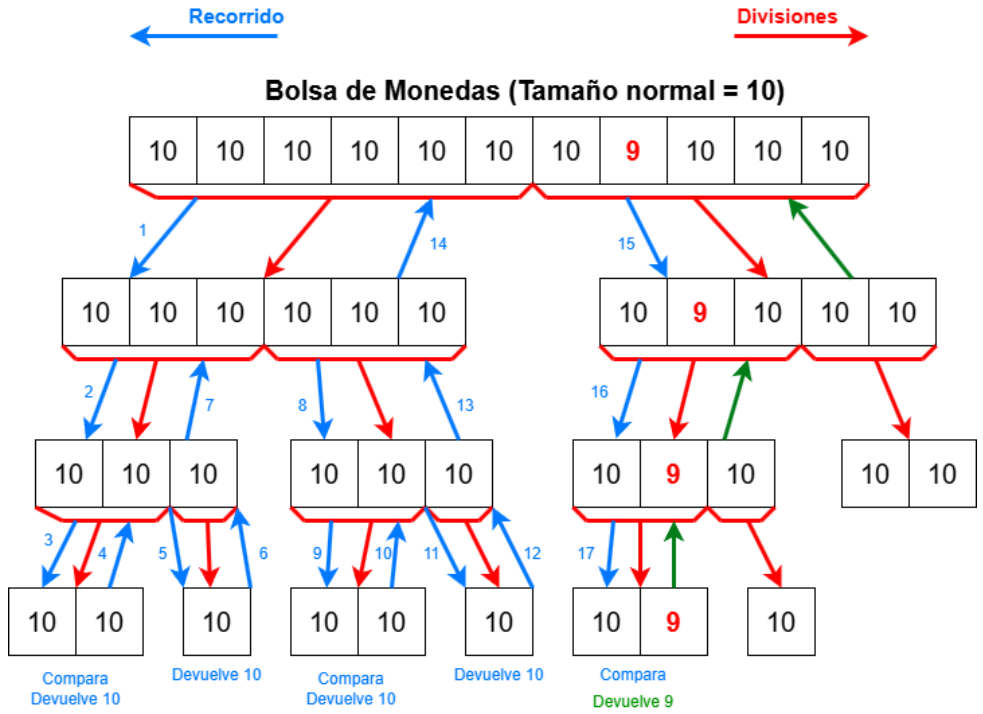
1.4. Seguimiento

El funcionamiento del algoritmo es el siguiente: se dispone de una **bolsa de monedas** de tamaño N, sobre la cual se itera dividiendo a la mitad la bolsa una y otra vez, reduciendo el problema hasta quedarse únicamente con una bolsa de tamaño 2, sobre las cuales se comparan ambas monedas y, en caso de que una de ellas sea de menor tamaño que la otra, se devolverá esta moneda diferente inmediatamente. En caso de que ambas monedas tienen el mismo tamaño, se retrocederá en la recursión un nivel para poder comparar el segundo par de monedas presente en la anterior división. A continuación, se dispone de un gráfico que muestra el funcionamiento del algoritmo.



Sin embargo, en caso de que se tenga que dividir una bolsa de monedas impares, se realiza el seguimiento solo que, al ser imposible comparar ambas monedas resultantes, simplemente la recursión devuelve la moneda en ese momento para compararla con la primer moneda presente en la segunda mitad de aquella división, y así determinar si se trata de una moneda diferente. Esto aplica para todos estos casos borde y, en caso de que la moneda impar se encuentre en el extremo de la segunda parte de la división, esta será devuelta ya que se trata

de la última moneda. Para este caso, también se dispone de un gráfico que ayuda a visualizar estas divisiones:



En cada uno de los graficos, cada flecha de color azul que indica el recorrido tiene un numero a su lado para poder identificar el orden del recorrido.

1.5. Complejidad

La ***Complejidad Temporal*** de este problema se puede calcular con el **Teorema Maestro**:

Ecuación de recurrencia: `T(n) = A T(n/b) + O(n^c)`

- **A**: Cantidad de llamadas recursivas
- **b**: Proporción en la que se dividen los subproblemas
- **c**: Costo de combinar los sub-resultados

En base al análisis de nuestra implementación, se pueden identificar las siguientes incógnitas:

- **A** = 2 (una llamada recursiva para cada mitad)
- **b** = 2 (siempre se divide el problema a la mitad)
- **c** = 1

Mediante el Teorema Maestro, se puede encontrar una solución para esta ecuación de recurrencia:

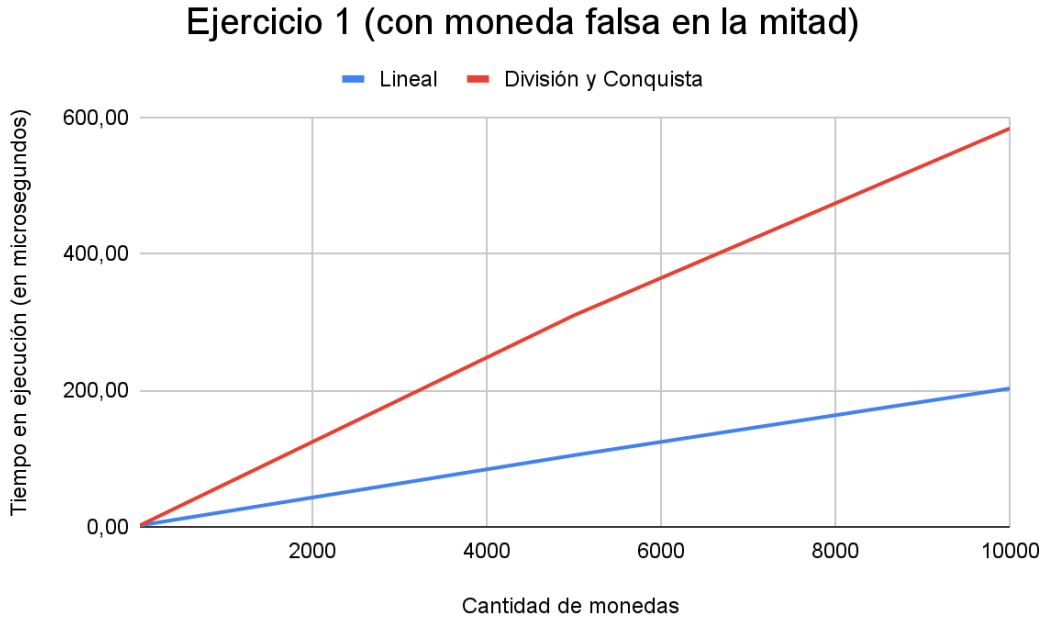
- Se identifica que `log_b (A) = c`
- En este caso, la solución es de la forma: `T(n) = O(n^c log_b n)`
- Entonces, la complejidad de este problema termina siendo: `T(n) = O(n log n)`

Por otro lado, la ***Complejidad Espacial*** de este problema permanece constante y siempre se trata de una lista de N elementos, representando el peso de cada moneda.

1.6. Sets de datos y Tiempos de Ejecución

Para este ejercicio se dispone el archivo `ejercicio_1_test.py` con los tests correspondientes para todos los sets de datos pedidos por el enunciado. También, se dispone del archivo `ejercicio_1_manual.py` el cual permite al usuario ejecutar tests personalizados y manejando bolsas de monedas de tamaños específicos, y ubicando a la moneda falsa manualmente.

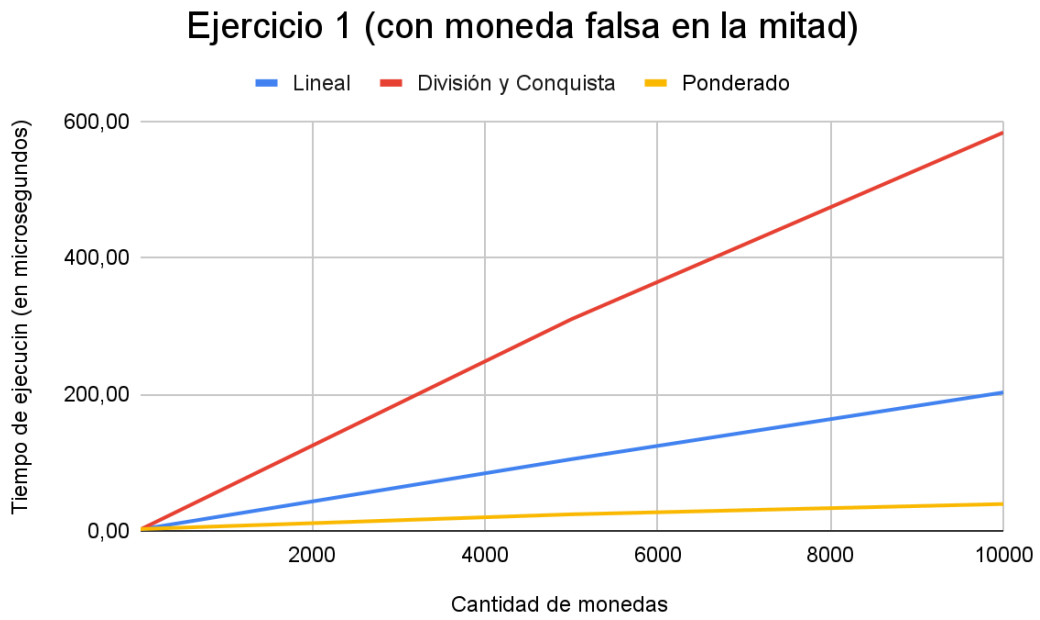
En base al resultado obtenido por los tests, se pueden diagramar los tiempos de ejecución para poder comparar un algoritmo de búsqueda lineal frente a nuestra implementación de División y Conquista. En este primer gráfico, se puede identificar como nuestra implementación cumple con la complejidad propuesta anteriormente: `T(n) = O(n log n)`



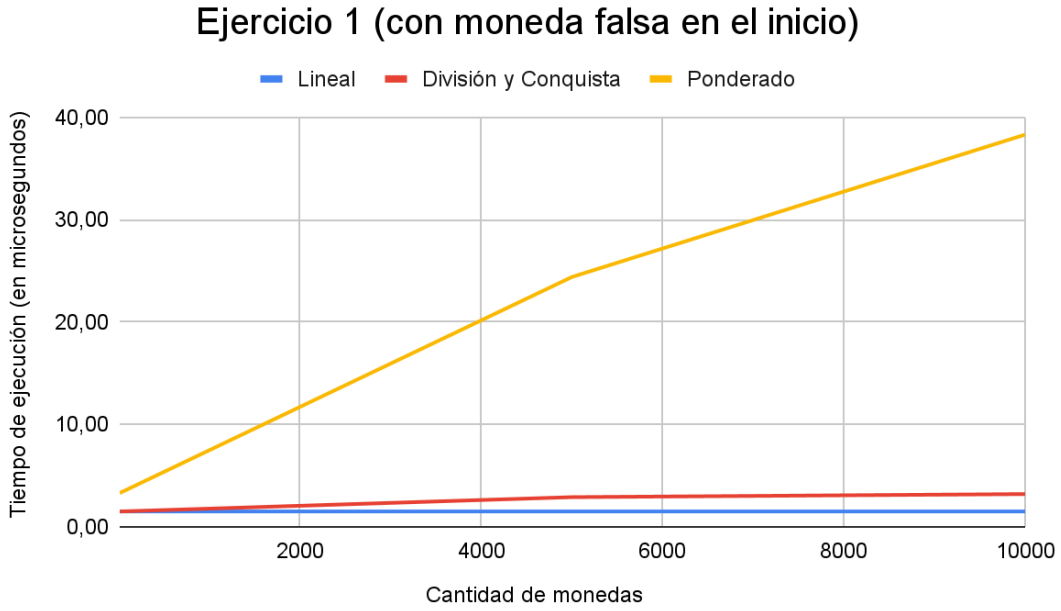
En base al análisis de los resultados de la ejecución de cada conjunto de datos (representando cada bolsa de N cantidad de monedas) se puede identificar como esta implementación de solución del problema, a comparación de una implementación lineal que simplemente recorre la bolsa de monedas hasta encontrar la falsa, termina siendo inferior en cuanto a rendimiento.

Es por esto que, al momento de haber diseñado nuestra implementación, habíamos logrado una implementación errónea de División y Conquista, buscando dividir la bolsa de monedas a la mitad y ponderando para encontrar por donde se encuentra la moneda falsa. Esta implementación fue conservada en los tests correspondientes de este ejercicio pero se destaca como un paso en nuestra investigación más no nuestro algoritmo final.

Siguiendo el análisis de los resultados de las ejecuciones, se puede identificar como aquella implementación errónea termina siendo ampliamente superior en cuanto a rendimiento, a pesar de tener que ponderar el peso total de cada mitad de bolsa de monedas. A continuación, se muestra un diagrama graficando justamente esto.



Por último, se muestra un último diagrama que permite identificar cómo se comporta cada algoritmo teniendo en cuenta una bolsa con una determinada cantidad de monedas, teniendo siempre a la moneda falsa al inicio de esta misma.



2. PROBLEMA 2

2.1. Enunciado y Supuestos

Existe un barrio-parque que podemos describir como una grilla de n x n manzanas. Aproximadamente, el 20% de las manzanas tiene un edificio. Una importante cadena de restaurantes ha detectado que no tiene ninguna sucursal instalada en el barrio. El plan de la cadena para posicionarse es construir la menor cantidad posible de restaurantes tal que que ninguno de los edificios quede a más de X manzanas de distancia del restaurante más cercano. Se pueden construir restaurantes en cualquier manzana, incluso en las que tienen edificios.

- Diseñar e implementar un algoritmo Greedy que resuelva este problema.
- Identificar claramente cuál es la regla greedy que utilizará el algoritmo.
- Optimalidad. ¿Es óptimo? Si lo es, demostrar. Si no lo es, dar un contraejemplo.
- NOTA: utilizar tamaños de datos con n = {10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000}.

También, tenemos en cuenta los siguientes supuestos:

- Sabiendo que grilla de n x n podría representar un mapa o un plano, suponemos que la posición de todos los edificios es conocida en primera instancia.
- El tamaño X de cobertura de manzanas es función del tamaño de la matriz (el 20% de n). Por ejemplo: si n es 5, X será 1, si n es 10, X será 2, si n es 100, X será 20…
- La cobertura es representada mediante la *distancia radial* por un cuadrado de lado X + 1, existen otras formas de representar la distancia, como por ejemplo la distancia Manhattan. A efectos prácticos, el código seguiría funcionando, pero se debería cambiar la implementación para adaptarse.

2.2. Diseño de la Solución

Una primera hipótesis que podemos tomar es que las potenciales mejores ubicaciones para colocar un restaurante se encuentran en las manzanas que contienen un edificio, pues, si construimos un restaurante en dichos sitios, nos aseguramos que en cualquier situación, nuestra construcción va a garantizar por lo menos una mínima cobertura (cobertura de un solo edificio).

En base a la hipótesis antes planteada, podemos generar nuestra primera estructura de datos: un diccionario cuya clave será una tupla (indicando la coordenada del edificio) y cuyo valor será un set de tuplas (indicando las coordenadas de los edificios que cubriría un restaurante en dicha ubicación, incluido el mismo edificio que estamos analizando).

De esta forma, podemos diseñar un algoritmo que tome decisiones en base a determinados óptimos locales (en nuestro caso, colocar un restaurante en la cuadra donde un edificio tenga una mayor cobertura de edificios) sin tener en cuenta un óptimo global.

2.3. Implementación del Algoritmo

Dentro de nuestra implementación, y siguiendo los pasos para diseñar nuestro algoritmo, hemos llegado a la siguiente función la cual se encargará de crear un diccionario con cada edificio y una lista ordenada (set) de cada uno de los edificios que están dentro de su radio de cobertura.

Luego, se itera sobre cada uno de los edificios para encontrar al edificio que tiene dentro una mayor cantidad de edificios sin cubrir. Una vez se encuentra al edificio con mayor cobertura, se coloca un restaurante en aquella posición y se dejan de tener en consideración a aquellos edificios que están dentro del rango de este restaurante.

```python
def construccion_de_restaurantes(edificios: list, tamano_barrio: int) -> list:

    restaurantes: list = list()
    diccionario_edificios: dict = crear_diccionario_edificios(edificios, tamano_barrio)

    while diccionario_edificios:
        max_cobertura: int = 0
        mejor_edificio: tuple = ()
        mejor_cobertura: set = set()

        for edificio, cobertura in diccionario_edificios.items():
            if len(cobertura) > max_cobertura:
                max_cobertura = len(cobertura)
                mejor_edificio = edificio
                mejor_cobertura = cobertura

        restaurantes.append(mejor_edificio)
        eliminar_edificios_cubiertos(diccionario_edificios, mejor_cobertura)

    return restaurantes
```

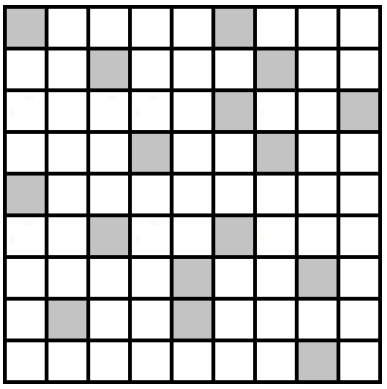
2.4. Seguimiento

La regla greedy que planteamos consiste en lo siguiente: en cada iteración del problema, buscaremos colocar un restaurante en la misma posición que se encuentre un edificio, priorizando siempre maximizar la cobertura del mismo. Esto potencialmente también reducirá la cantidad de restaurantes necesarios para cubrir todos los edificios presentes en la grilla.

De esta forma, en cada iteración estamos buscando por un óptimo local con la esperanza de alcanzar un óptimo global (aunque esto no esté asegurado).

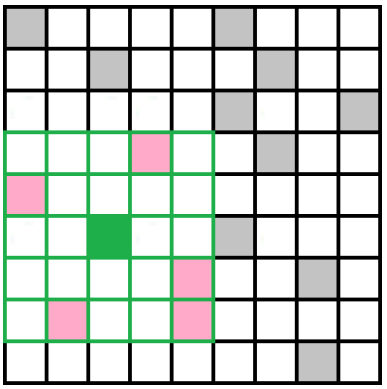
A continuación, se determina un seguimiento correspondiente para poder visualizar el algoritmo en funcionamiento:

- Se utiliza una instancia de un barrio de tamaño 9 x 9:

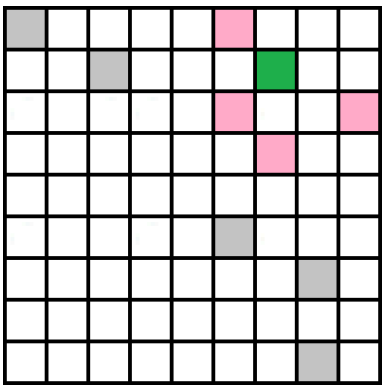


En primera instancia, tenemos esta grilla, las casillas en gris representan edificios. Se puede observar que hay 16 de ellos ($9^9 \cdot 0{,}20 = 16{,}2$ aplicando redondeo = 16). El resto de casillas en blanco representan manzanas sin edificios. El rango de cobertura en este caso será de 2 ($0{,}2 \cdot 9 = 1{,}8$ aplicando redondeo = 2).

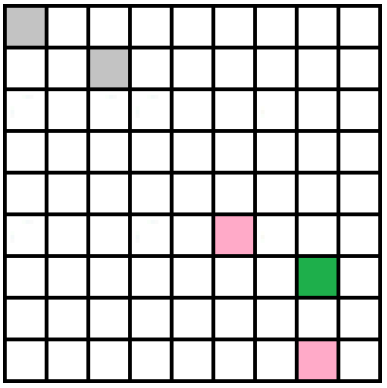
- El primer paso es analizar en cada edificio, y trazar un área imaginaria, la cual representará el radio de cobertura en caso de elegir construir un restaurante en dicha posición, siguiendo esta lógica, el edificio que tomaremos será el que está en la posición (5, 2), se verá mejor a continuación:



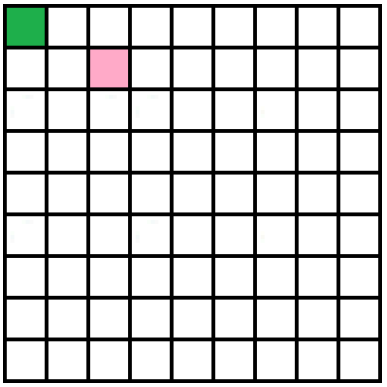
Se observa como se marca el edificio anteriormente indicado, también se observa el área alrededor del mismo, que indica el rango que tendrá el restaurante al ubicarlo en dicha posición, como es el conjunto que más cubre (**IMPORTANTE:** notar que si se elige el edificio en (5, 5) también se cubrirán 6 edificios, pero como se recorren los edificios en orden (de izquierda a derecha y de arriba a abajo) no lo tomaremos, pues aunque su cobertura sea la misma, aparece posteriormente en la grilla).



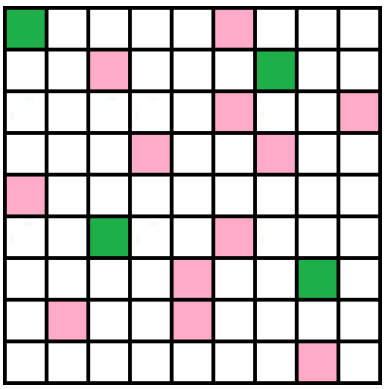
Se observa como los edificios que ya fueron marcados anteriormente, ya fueron removidos de la grilla. En la siguiente iteración, seleccionamos el edificio (1, 6), pues si trazamos un área alrededor de él, veremos como toma, en total, 5 edificios.



En la anteúltima iteración, seleccionamos el edificio (6, 7), siguiendo la lógica de antes, nos ofrece una mayor cobertura.



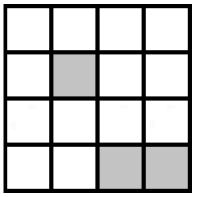
En la última iteración, seleccionamos el edificio (0, 0), notar que, en este caso, podíamos seleccionar cualquier edificio, ya que el resultado no iba a cambiar, sin embargo, con el criterio anterior, seleccionamos el de primera aparición.



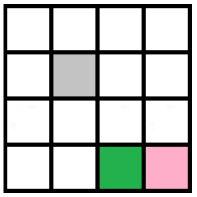
- **Resultado final:** observamos que no existen edificios sin cobertura, y en este caso, nos dio el óptimo, pues no se puede completar esta grilla con un número menor a 4 restaurantes.

Es muy importante notar que si bien este algoritmo nos da soluciones factibles, **NO SIEMPRE** nos dará la solución óptima. Podemos reducir el problema 2 de este trabajo práctico a **set cover**. Es sabido que no se conoce un algoritmo óptimo que se ejecute en tiempo polinomial que resuelva este problema. Sin embargo, la solución a través del algoritmo greedy, si bien no siempre dará un resultado óptimo, es una excelente aproximación y sabemos que, en general dará resultados factibles, que no se alejan mucho del óptimo.

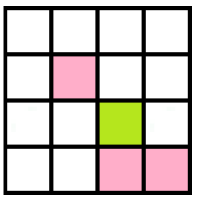
Como el algoritmo no es óptimo, lo podemos probar con un contraejemplo que veremos a continuación:



- Tenemos esta nueva instancia del problema, en este caso la grilla es de 4 x 4, por lo que hay 3 edificios ($4^4 \cdot 0{,}2 = 3{,}2$ aplicando redondeo = 3) el rango que tomaremos, será por su parte de 1 ($4 \cdot 0{,}2 = 0{,}8$, aplicando redondeo = 1)



Siguiendo el algoritmo antes descrito, es casi instantáneo notar que tomaremos el edificio en (3, 2), ya que aparece primero y nos garantiza una cobertura de 2. Ahora, tenemos un problema… Si o si vamos a necesitar otro restaurante para cubrir el edificio restante, entonces utilizaremos finalmente 2 restaurantes. Si hubiésemos elegido colocar un restaurante en (2, 2) el tablero estaría completamente cubierto, únicamente utilizando 1 restaurante.



Solución óptima: se observa como si colocamos un restaurante en la posición antes indicada, solamente requerimos de uno de ellos. Esta solución es mejor que la que plantea el algoritmo greedy propuesto.

2.5. Complejidad

La ***complejidad temporal*** del algoritmo propuesto será fuertemente dependiente de la cantidad de edificios que tengamos. Denotando como K al número de edificios presentes en una grilla, sabemos que $K = 0.20 \cdot n^2$ por lo que una complejidad de O(K) en realidad puede ser reescrita como $O(n^2)$ (las constantes no nos modifican las cotas asintóticas superiores).

- Analizando la función **calcular_cobertura()**, la misma recorre todos los edificios, su complejidad es de O(K), el resto de operaciones en esta función es O(1).
  - **Complejidad resultante:** O(K).
  - Analizando la función **crear_diccionario_edificios()**, vemos que la misma presenta un ordenamiento y también recorre todos los edificios. A su vez, hace uso de la función **calcular_cobertura**() dentro del ciclo. La complejidad en este caso será de $O(K \cdot \log_2(K))$ para el ordenamiento. El doble ciclo anidado tiene una complejidad de $O(K) \cdot O(K) = O(K^2)$.
  - **Complejidad resultante:** $O(K \cdot \log_2(K)) + O(K^2) = O(K^2)$.
- Analizando la función **eliminar_edificios_cubiertos()**, el primer ciclo itera sobre el set de mayor cobertura, si dicho set tiene M elementos, la complejidad será de O(M), la operación de eliminación en un diccionario es O(1). Como M puede tomar valores entre 1 y K. Si M = 1 el ciclo tendrá una complejidad de O(1), si M = K, el ciclo tendrá una complejidad de O(K).

El segundo ciclo itera sobre un diccionario de edificios ya reducido, con K - M elementos. Luego se realiza una operación de diferencia de conjuntos la cual tiene una complejidad de O(M), siendo M la cantidad de elementos del set (los valores del diccionario). La complejidad entonces será de $O(K - M) \cdot O(M) = O(K - M \cdot M)$. Como M puede tomar valores entre 1 y K, si M es 1 la complejidad será de O(K), si M es K, el primer ciclo eliminará el diccionario por completo y el segundo ciclo no se va a ejecutar nunca.

  - **Complejidad resultante:** Es muy importante notar que aunque haya dos ciclos, ambos se “contrarrestan” entre sí. La mayor complejidad posible que puede otorgar esta función es en realidad O(K).
- Analizando la **función principal** del código, primero hace uso de la función crear_diccionario_edificios() la cual ya analizamos tiene una complejidad de $O(K^2)$. El primer ciclo while tiene una complejidad de O(K), el ciclo for por su parte también tiene una complejidad de O(K). La función eliminar_edificios_cubiertos() ya fue analizada con

anterioridad, la misma tiene una complejidad de O(K). El resto de operaciones de esta función tiene una complejidad de O(1).

Complejidad final: $O(K) + O(K) \cdot (O(K) + O(K)) = O(K^2) = O(n^4)$

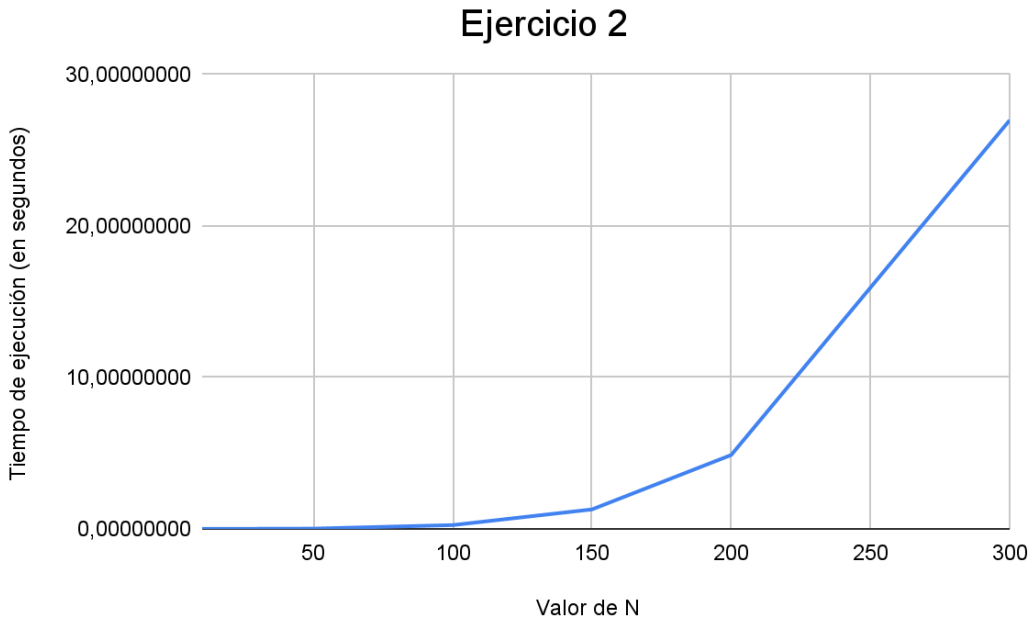
- La ***complejidad espacial*** por su parte también será muy dependiente de la cantidad de edificios. La estructura de datos más robusta en todo el algoritmo es el diccionario.
  - Al calcular la cobertura de un edificio, vemos que el peor escenario posible para la complejidad espacial será que un edificio cubra a su vez a todos los edificios de la grilla. En este caso, el set de cobertura tendrá una longitud K, por consiguiente, la complejidad espacial será O(K).
  - Al crear el diccionario, en el peor escenario posible existen K edificios de los cuales todos están cubiertos entre sí. En este escenario, tendremos un conjunto de longitud K por cada edificio en el diccionario, brindando una complejidad espacial de $O(K^2)$.
  - Otro escenario posible es que ningún edificio tenga en cobertura a otro, o sea, que todos los edificios de la grilla tengan la mínima cobertura posible. En este caso, la lista de restaurantes se llenará con K elementos y tendrá una complejidad espacial de O(K).

Complejidad resultante: El peor escenario es en el cual tenemos K edificios con un set de cobertura de K por cada uno. En este caso, la complejidad espacial será de $O(K^2) = O(n^4)$

2.6. Sets de datos y Tiempos de Ejecución

Al igual que para el Ejercicio 1, en este caso también se dispone del archivo `ejercicio_2_test.py` con tests correspondientes con tipos de datos en orden logarítmico. También, se dispone del archivo `ejercicio_1_manual.py` el cual permite que el usuario pueda ejecutar una grilla de N x N cantidad de edificios totalmente personalizada, pudiendo elegir sus dimensiones y colocar edificios en los lugares que crea necesario, para luego poder correr el generador de restaurantes y ver en un archivo .csv la disposición de los edificios y, en otro, donde se posicionaron los restaurantes.

Teniendo estos métodos de testeo de nuestro algoritmos a disposición, se puede identificar la complejidad final correspondiente a **$O(N^4)$** debido a que, en el momento de incrementar N correspondiente a una de las dimensiones de la grilla de cuadras, la cantidad de edificios escala considerablemente, manteniéndose siempre como el 20%, lo que permite identificar un incremento muy considerable en cuanto a tiempo de ejecución a medida que aumenta el N, llegando a plazos de tiempo muy largos en caso de que N sea mayor a 400.



En este gráfico se pueden identificar los tiempos de ejecución más considerables que hemos logrado registrar de forma manual en nuestro algoritmo, con una planilla de hasta 300 x 300 cuadras.

3. PROBLEMA 3

3.1. Enunciado y Supuestos

Desarrollar un algoritmo que construya un cuadrado mágico de nxn. Un cuadrado mágico es aquél en que cada celda tiene un número de 1 a n^2, que no se repite en otras celdas, y en el cual la suma de los números en su diagonal principal coincide con la suma de los números de cada fila y columna. Implementarlo usando backtracking.

- NOTA: utilizar tamaños de datos con n = {5, 6, 7, 8, 9, 10, 11, 12}.

También, tenemos en cuenta los siguientes supuestos:
- El cuadrado más pequeño necesario que el algoritmo pueda construir es uno de 3 x 3
- El problema tiene como requerimiento que la suma de elementos de una fila, columna o la diagonal principal de lo mismo. Pero la solución toma la definición de cuadrado mágico en la cual la diagonal secundaria también debe de cumplir la condición.
- Al sumar los elementos de cualquier fila, columna o diagonal principal o secundaria debe dar una constante mágica que sigue la fórmula: **$n(n^2 + 1)/2$**

3.2. Diseño de la Solución

El algoritmo recorre las celdas a rellenar y en cada una prueba las opciones posibles utilizando validaciones y aplicando backtracking para descartar soluciones parciales las cuales se puede asegurar que no son capaces de encontrar una solución final. Se utilizan los siguientes Objetos:
- PossibleValue, para poder almacenar el valor y un booleano para saber si el valor fue usado o no.
- PossibleValues, para poder almacenar los valores posibles. Además lleva un contador de cuantos valores disponibles hay el cual es modificado al utilizar sus métodos que marcan un valor como usado o no usado.
- CachedRowColumnDiagonalCalculations, para guardar las sumas de las filas, columnas y las diagonales principal y secundaria. Además también calcula y almacena la constante mágica.

El algoritmo empieza con el método `buildMagicSquare()` el cual prepara los datos iniciales a partir del N ingresado e invoca al método fill para llenar el cuadrado.
- El método `fill()` es el encargado de llenar el cuadrado mágico. Este itera los valores posibles y en caso de encontrar valores válidos. Es decir que sean compatibles y no hayan sido usados los guarda como solución parcial. Busca una solución completa invocando de nuevo este método. En caso de no encontrar una solución completa se descarta la solución parcial, se marca el valor como no usado y se sigue con la siguiente opción.
- El `método compatible()` se encarga de verificar si el nuevo valor es compatible a partir del resultado sumándose a las filas, columnas y diagonales y revisando que la suma no supere la constante mágica. O en caso de que se haya completado una fila, columna o diagonal que la suma sea igual a la constante mágica.

3.3. Implementación del Algoritmo

Implementación del método `fill()` para el llenado de celdas:

```
def fill(square, sideSize, cachedRowColumnDiagonalCalculations, possibleValues, row, col):
   # Funcion recursiva de Backtracking
   values = possibleValues.getValues()
   for possibleValue in values:

       if len(square) == row and len(square) < sideSize:
           square.append([])
       if not possibleValue.used and compatible(cachedRowColumnDiagonalCalculations, row, col,
  possibleValue.value):
           cachedRowColumnDiagonalCalculations.addItem(row, col, possibleValue.value)
           if len(square[row]) == col:
               square[row].append(possibleValue.value)
           else:
               square[row][col] = possibleValue.value
           possibleValues.markUsed(possibleValue)

           nextRow = 0
           nextCol = 0
           if col == sideSize - 1:
               nextRow = row + 1
               nextCol = 0
           else:
               nextRow = row
               nextCol += col + 1
           if possibleValues.getAvailablesCount() == 0:
               return True

           solutionExists = fill(square, sideSize, cachedRowColumnDiagonalCalculations,
  possibleValues, nextRow, nextCol)
           if solutionExists:
               return True
           else:
               cachedRowColumnDiagonalCalculations.removeItem(row, col, possibleValue.value)
               possibleValues.markUnused(possibleValue)
               square[row][col] = 0
               if possibleValues.getAvailablesCount() == 0:
                   return False
   return False
```

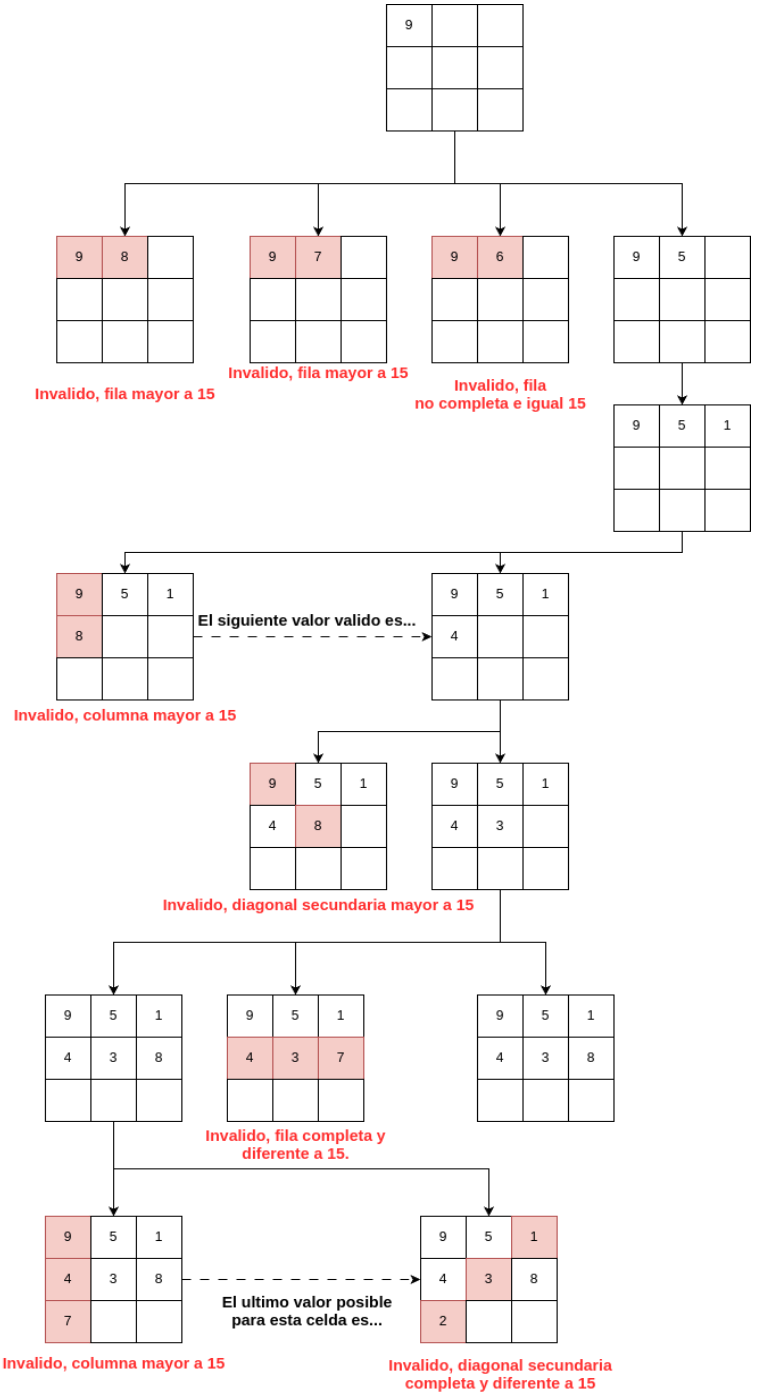
Implementación del método `compatible()` para el análisis de compatibilidad:

```
def compatible(cachedRowColumnDiagonalCalculations, row, col, value):
    magicConstant = cachedRowColumnDiagonalCalculations.magicConstant
    # COMPARA LAS SUMAS DE FILA, COLUMNA Y DIAGONALES
    rowTotal = cachedRowColumnDiagonalCalculations.getRowTotal(row) + value
    # SI LA SUMA DE LA FILA ES MAYOR A LA CONSTANTE MÁGICA, NO ES COMPATIBLE
    if rowTotal > magicConstant:
        return False
    # SI LA SUMA DE LA COLUMNA ES MAYOR A LA CONSTANTE MÁGICA, NO ES COMPATIBLE
    columnTotal = cachedRowColumnDiagonalCalculations.getColumnTotal(col) + value
    if columnTotal > magicConstant:
        return False
    # SI LA SUMA DE LA DIAGONAL PRINCIPAL ES MAYOR A LA CONSTANTE MÁGICA, NO ES COMPATIBLE
    mainDiagonalTotal = cachedRowColumnDiagonalCalculations.getMainDiagonalTotal() + value
    if row == col and mainDiagonalTotal > magicConstant:
        return False
    # SI LA SUMA DE LA DIAGONAL SECUNDARIA ES MAYOR A LA CONSTANTE MÁGICA, NO ES COMPATIBLE
    secondaryDiagonalTotal = cachedRowColumnDiagonalCalculations.getSecondaryDiagonalTotal() + value
    if row + col == cachedRowColumnDiagonalCalculations.sideSize - 1 and secondaryDiagonalTotal >
  magicConstant:
        return False
    # VALIDA QUE SI ES LA ULTIMA CELDA DE FILA DE LA CONSTANTE MÁGICA
    isLastCellInRow = col == cachedRowColumnDiagonalCalculations.sideSize - 1
    if isLastCellInRow and rowTotal != magicConstant:
        return False
    # VALIDA QUE SI ES LA ULTIMA CELDA DE COLUMNA DE LA CONSTANTE MÁGICA
    isLastCellInColumn = row == cachedRowColumnDiagonalCalculations.sideSize - 1
    if isLastCellInColumn and columnTotal != magicConstant:
        return False
    # VALIDA QUE SI ES LA ULTIMA CELDA DE LA DIAGONAL SECUNDARIA DE LA CONSTANTE MÁGICA
    isLastCellInSecondaryDiagonal = col == 0 and row == cachedRowColumnDiagonalCalculations.sideSize
  - 1
    if isLastCellInSecondaryDiagonal and secondaryDiagonalTotal != magicConstant:
        return False
    # VALIDA QUE SI ES LA ULTIMA CELDA DE LA DIAGONAL PRINCIPAL DE LA CONSTANTE MÁGICA
    isLastCellInMainDiagonal = row == col and row == cachedRowColumnDiagonalCalculations.sideSize - 1
    if isLastCellInMainDiagonal and mainDiagonalTotal != magicConstant:
        return False
    return True
```

3.4. Seguimiento

Hacer un seguimiento de un algoritmo que utiliza *backtracking* puede ser algo complejo. Para facilitar esta tarea, podemos pensar en el árbol de decisiones, en el cual la raíz será la matriz vacía, luego iremos construyendo soluciones parciales en base a una rama del árbol. Si vemos que ya no podremos tener soluciones válidas, podamos la rama entera.

Para simplificar la tarea del seguimiento del código, a continuación se encuentra una imagen tratando de simular lo que haría el algoritmo en una rama del árbol de decisiones.

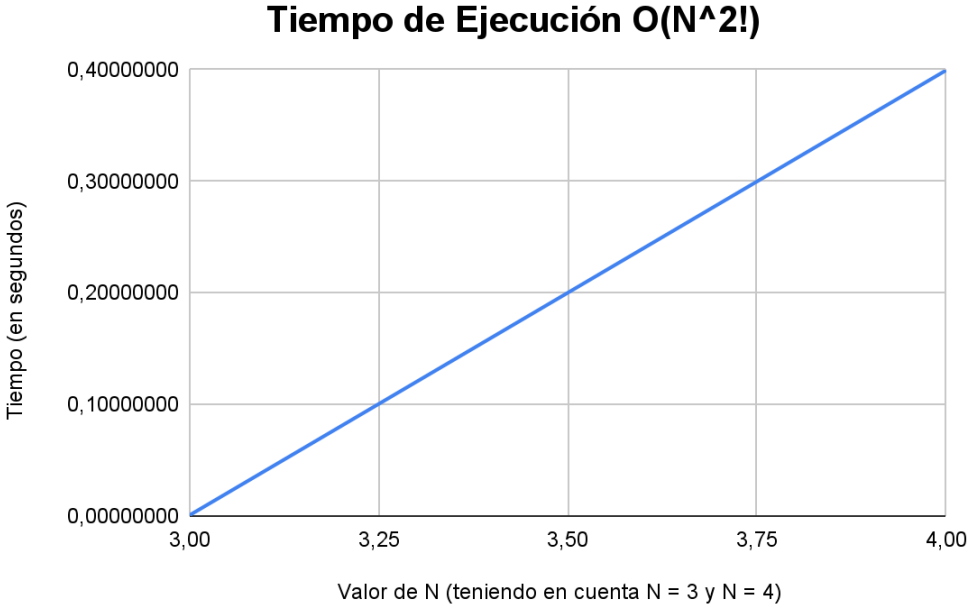


3.5. Complejidad

- El método `buildMagicSquare()` Arma un array de valores posibles lo cual tiene una complejidad de O(n²).
- El método `fill()` recorre las celdas del cuadrado y por cada celda, en el peor de los casos, se recorren los n² elementos posibles con los cuales se puede llenar esa celda y con cada uno de estos valores se invoca recursivamente a `fill()`. A pesar de que se puedan realizar podas gracias al backtracking la complejidad sería la misma a la solución por fuerza bruta. La cual sería **O(n²!)**.
- El método `compatible()` tiene complejidad O(1) ya que al solo realizar sumas y comparaciones no varía dependiendo del tamaño del input.
- Los métodos `addItem()` y `removeItem()` solo suman o restan el valor agregado/removido y no dependen del tamaño del input por lo cual tienen complejidad de **O(1)**.
- Con N grande la complejidad del algoritmo queda como **O(n²!)** ya que crece mucho más rápido.

3.6. Sets de datos y Tiempos de Ejecución

No es necesario un set de datos complejos para probar la aplicación ya que construye los datos necesarios a partir del N.Es por esto que, al igual que los demás ejercicios, se dispone del archivo `ejercicio_3_test.py` para poder poner a prueba esta implementación con un N hasta 4, debido a que la complejidad temporal incrementa de tal forma que, al llegar a N=5, se excede por mucho tiempo.



En este gráfico se puede identificar el rápido crecimiento que tienen los tiempos de ejecución al aumentar mínimamente el N, comenzando desde un N = 3.

3.7. Algoritmo Alternativo Propuesto

Para mejorar nuestra implementación del algoritmo, hemos planteado el diseño de un algoritmo de Backtracking recursivo, en el cual se van calculando incrementalmente matrices de tamaños más pequeños, hasta llegar al tamaño deseado siempre siguiendo una regla explicada a continuación.

Matriz de 3 x 3

- Tiene una disposición de números del 1 al 9
- **Constante mágica** = 15.
  - La constante mágica de la matriz se puede calcular mediante la siguiente operación: $\frac{n \cdot (n^2+1)}{2} = \frac{3 \cdot (3^2+1)}{2} = 15$
  - Esto es equivalente a sumar todos los números del 1 al n^2 y dividirlos por el tamaño de la matriz.

Matriz de 4 x 4

- Tiene una disposición de números del 1 al 16
- **Constante mágica** = 34.
  - La constante mágica de la matriz se puede calcular mediante la siguiente operación: $\frac{n \cdot (n^2+1)}{2} = \frac{4 \cdot (4^2+1)}{2} = 34$
  - Esto es equivalente a sumar todos los números del 1 al n^2 y dividirlos por el tamaño de la matriz.

Matriz de 5 x 5

- Tiene una disposición de números del 1 al 25
- **Constante mágica** = 65
- Se puede formar utilizando una matriz de 3 x 3 que debe cumplir ciertos requisitos:
  - Los elementos de dicha matriz deben ser del $(5^2 - 3^2) / 2 + 1 =$ **9** al $(5^2 - 3^2) / 2 + 3^2 =$ **17.**
  - La constante mágica se calcula como la sumatoria de todos los elementos divididos 3: (9 + 10 + ... + 17) / 3 = 39
- Para los elementos restantes:
  - Tenemos la primera mitad de números: 1, 2, 3, 4, 5, 6, 7, 8.
  - Tenemos la segunda mitad de números: 18, 19, 20, 21, 22, 23, 24, 25.
  - Invertimos la segunda mitad: 25, 24, 23, 22, 21, 20, 19, 18.
- Ahora podemos agrupar los números en tuplas de a 2 de la siguiente manera:
  - **(**(1, 25), (2, 24), (3, 23), ..., (8, 18)) Notar que todos estos números suman $5^2 + 1 = 26$.
  - De todas las tuplas, usaremos 5 + 3 de ellas para la matriz de 5 x 5 (o sea, todas).

Matriz de 6 x 6

- Tiene una disposición de números del 1 al 36
- **Constante mágica** = 111
- Se puede formar utilizando una matriz de 4 x 4 que debe cumplir ciertos requisitos:
  - Los elementos de dicha matriz deben ser del $(6^2 - 4^2) / 2 + 1 =$ **11** al $(6^2 - 4^2) / 2 + 4^2 =$ **26.**
  - La constante mágica se calcula como la sumatoria de todos los elementos divididos 4: $(11 + 12 + ... + 26) / 4 = 74$
- Para los elementos restantes:
  - Tenemos la primera mitad: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
  - Tenemos la segunda mitad: 27, 28, 29, 30, 31, 32, 33, 34, 35, 36.
  - Invertimos la segunda mitad: 36, 35, 34, 33, 32, 31, 30, 29, 28, 27.
- Ahora podemos agrupar los números en tuplas de a 2 de la siguiente manera:
  - ((1, 36), (2, 35), (3, 34), ..., (10, 27)) Notar que todos estos números suman $6^2 + 1$.
  - De todas las tuplas, usaremos 6 + 4 de ellas para la matriz de 6 x 6 (o sea, todas).

Matriz de 7 x 7

- Tiene una disposición de números del 1 al 49
- **Constante mágica** = 175
- Se puede formar utilizando una matriz de 3 x 3 y una matriz de 5x5, que deben cumplir ciertos requisitos:
- Para la matriz de 3 x 3:
  - Los elementos de dicha matriz deben ser del $(7^2 - 3^2) / 2 + 1 =$ **21** al $(7^2 - 3^2) / 2 + 3^2 =$ **29.**
  - La constante mágica se calcula como la sumatoria de todos los elementos dividido 3: $(21 + 22 + ... + 29) / 3 = 75$
- Para los elementos restantes:
  - Tenemos la primera mitad de números: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 20.
  - Tenemos la segunda mitad de números: 30, 31, 32, 33, 34, 35, 36, ..., 49.
  - Invertimos la segunda mitad: 49, 48, 47, 46, 45, 44, 43...
- Ahora podemos agrupar los números en tuplas de a 2 de la siguiente manera:
  - ((1, 49), (2, 48), (3, 47), ..., (20, 30)) Notar que todos estos números suman $7^2 + 1$.
  - De todas las tuplas, usaremos 7 + 5 de ellas para la matriz de 7 x 7 (Las primeras 12).
  - De todas las tuplas, usaremos 5 + 3 de ellas para la matriz de 5 x 5 (Las últimas 8).

A continuación, se muestra un ejemplo de visualización del cuadrado mágico producto del algoritmo de backtracking propuesto, con N = 7:

49	48	11	46	6	12	3
7	13	14	31	32	35	43
8	30	28	21	26	20	42
45	33	23	25	27	17	5
9	34	24	29	22	16	41
10	15	36	19	18	37	40
47	2	39	4	44	38	1

Siguiendo la lógica antes planteada, en esta imagen se ve el cuadrado de 3 x 3 interno con valores del 21 al 29, cuya constante mágica es 75. Luego, en la capa de la matriz de 5 x 5 se observa que están los pares incluidos en las tuplas ((13, 37), (14, 36), (15, 35), (16, 34), (17, 33), (18, 32), (19, 31), (20, 30)) y que si sumamos cualquier tupla de números nos dará como resultado $7^2 + 1$.

Como último, podemos observar la capa más exterior, la cual completa la matriz de 7 x 7 y en la cual se incluyen las tuplas ((1, 49), (2, 48), (3, 47), ((4, 46), (5, 45), (6, 44), (7, 43), (8, 42), (9, 41), (10, 40), (11, 39), (12, 38)), notar que nuevamente, si sumamos cualquier tupla de números también nos dará como resultado $7^2 + 1$.

Este mismo procedimiento, aplica a cualquier matriz de tamaño mayor o igual a 5, pues si tuviésemos que implementar una matriz de, por ejemplo, 16x16, tendríamos que hacer una construcción progresiva de matrices. Empezamos por una de 4 x 4, luego 6 x 6, luego 8 x 8, luego 10 x 10, luego 12 x 12, luego 14 x 14 y finalmente 16 x 16. Solamente tendremos que tener especial cuidado con la paridad del n: si n es par, deberemos construir como base una matriz de 4 x 4, de lo contrario, la matriz base debe ser de 3 x 3.

¿Por qué es backtracking? Para empezar, la matriz "núcleo" ya sea de 3 x 3 o de 4 x 4 se calcula mediante backtracking. Ahora, para cada matriz de mayor tamaño, si bien tenemos una serie de tuplas definidas a colocar, estas mismas no pueden ser dispuestas de cualquier manera. Pues, recordemos que estamos sujetos a restricciones propias del cuadrado mágico como lo pueden ser no excederse o no quedarse por debajo de cierto valor. Entonces, deberíamos probar distintas permutaciones. En el peor caso, haremos la misma cantidad de permutaciones que de tuplas disponibles.

Teniendo en cuenta este algoritmo propuesto, la **complejidad temporal** resultante es ampliamente inferior a **$O(N^2!)$**, tendiendo a **$O(N!)$** mientras puede cumplir con el enunciado al poder ser aplicado mediante *Backtracking*.

REFERENCIAS:

Documentación de Python. (2023). *Complejidad temporal.* [TimeComplexity - Python Wiki]

Graham Cormode, Howard Karloff, Anthony Wirth. (2010). *Set Cover Algorithms For Very Large Datasets*. AT&T Labs–Research.

Petr Slavík. (1996). *A Tight Analysis of the Greedy Algorithm for Set Cover*. Department of Mathematics, State University of New York at Buffalo.

Peter Loly, Ian Cameron, Walter Trump, Daniel Schindel. (2008). *Magic square spectra.* Department of Physics and Astronomy, The University of Manitoba, Winnipeg, Manitoba, Canada [R3T2N2]. Gymnasium Stein, Faber-Castell-Allee 10, D-90547 Stein, Germany. Department of Physics and Astronomy, University of Western Ontario, London, Ontario, Canada [N6A3K7].

Weisstein, Eric W. (2024). *Magic Square*. MathWorld. [Magic Square -- from Wolfram MathWorld]