INFORME PROYECTO I - ADA II

Víctor Manuel Hernández Ortiz *2259520*
Nicolás Mauricio Rojas Mendoza *2259460*
Esteban Alexander Revelo Salazar *2067507*

I. LA TERMINAL INTELIGENTE

A. *Entender el problema*

Para comprender el problema, se pide completar la transformación de dos problemas de la terminal, cada uno mediante dos soluciones distintas.

Transformando **ingenioso** a **ingeniero**, dos soluciones

Operación	Cadena
advance	ingenioso
advance	ingenioso
advance	ingenioso
advance	ingenioso
advance	ingenioso
advance	ingenioso
replace with e	ingenieso
replace with r	ingeniero

El costo de la anterior solución es de $6a + 2r$

Operación	Cadena
advance	ingenioso
advance	ingenioso
advance	ingenioso
advance	ingenioso
advance	ingenioso
advance	ingenioso
kill	ingeni
insert e	ingenie
insert r	ingenier
insert o	ingeniero

El costo de la anterior solución es de $6a + 3i + k$

Transformando **francesa** a **ancestro**, dos soluciones

Operación	Cadena
replace with a	arancesa
replace with n	anancesa
replace with c	ancncesa
replace with e	ancecesa
replace with s	ancesesa
replace with t	ancestsa
replace with r	ancestra
replace with o	ancestro

El costo de la anterior solución es de $8r$

Operación	Cadena
delete	rancesa
delete	ancesa
advance	ancesa
advance	ancesa
advance	ancesa
advance	ancesa
advance	ancesa
kill	ances
insert t	ancest
insert r	ancestr
insert o	ancestro

El costo de la anterior solución es de $5a + 2d + 3i + k$

B. *Caracterizar la estructura de la solución óptima*

La solución al problema de edicion de transformación de cadenas se implementa en un enfonque de programación Dinámica para calcular el costo mínimo de transformar una cadena de caracteres X en otra cadena Y considerando las operaciones de avanzar, borrar ,insertar, reemplazar y kill.

Dentro de lo que es la programación dinámica tenemos lo que es la construccion de la matriz de soluciones M, la cual se utiliza para almacenar los costos mínimos acumulados para transformar los primeros **i** caracteres de **X** en los primeros **j** caracteres de **Y**.

C. *Definir recursivamente el valor de la solución óptima*

Supongamos las cadenas X=”abc” y Y=”adc”, con los costos dados: a=1, d=2, r=3, i=2i , k=1. Entonces, Inicializamos la matriz

```
M[0][0]=0,
M[i][1]=i*2 (eliminaciones),
M[0][j]=j*2 (inserciones).
```

i/j	0	1	2	3
0	0	2	4	6
1	2			
2	4			
3	6			

llenamos la matriz usando la recurrencia

1. Celda M[1][1]:

```
Comparar X[0]=a con Y[0]=a
    Avanzar M[0][0] + 1 = 1
Minimo M[1][1] = 1
```

2. Celda M[1][2]:

```
comparar X[0]=a con Y[1]=d
    insertar M[1][1] + 2 = 3
    reemplazar M[0][1] + 3 = 5
Minimo M[1][2] = 3
```

3. Celda M[2][3]:

```
comparar X[1]=b con Y[2]=c
    reemplazar  M[1][2] + 3 = 6
    eliminar M[2][2] + 2 = 5
Minimo M[2][3] = 6
```

Matriz final:

i/j	0	1	2	3
0	0	2	4	6
1	2	1	3	5
2	4	3	3	6
3	6	5	5	6

El costo mínimo para transformar X=”abc” en Y=”adc” es M[3][3]=6.

D. Calcular el valor de una solución óptima

La solución optima se construye con los siguientes pasos.

1. Inicialización: Se crea una matriz M de tamaño (m+1).(n+1), donde m=X y n=Y. La matriz se inicializa con valores infinitos para garantizar que las actualizaciones posteriores consideren siempre el costo más bajo.

- **Costos para transformar una cadena no vacía en una vacía:**
  - Para la primera columna (j=0), el costo se calcula como M[i][0]=i x costo['delete'], ya que para eliminar todos los caracteres de X, solo se realizan operaciones de eliminación.
- **Costos para transformar una cadena vacía en otra no vacía**
  - la primera fila (i=0), el costo se calcula como M[0][j]=j x costo['insert'], ya que para formar Y desde una cadena vacia, solo se realizan operaciones de inserción.

2. Calcular la solucion en base al llenado de la matriz: para cada celda M[i][j], se evaluan los costos asociados a las operaciones y se selecciona el mínimo:

- **Costo de avanzar:**
  - Si X[i-1] = Y[j-1], el costo es M[i-1][j-1]+ costo['advance']. Esto indica que los caracteres X[i] y Y[j] son iguales y no se requiere ninguna operación.
- **Costo de borrar:**
  - El costo es M[i-1][j] + costo['delete'], ya que se elimina X[i-1].
- **Costo de insertar:**
  - El costo de insertar es M[i][j-1] + costo['insert']. Esto inserta Y[j-1] en X.
- **Costo de reemplazar:**
  - Si X[i-1] != Y[j-1],el costo es M[i-1][j-1] + costo['replace'], ya que se reemplaza X[i-1] por Y[j-1].
- **Costo de kill:**
  - Si estamos al final de la cadena Y (j=n), se evalua M[i][n] = min(M[i][n], costo ['kill']), que representa eliminar el resto de los caracteres de X.

3. Recuperación de las operaciones: Una vez calculada la matriz M, se reconstruye la secuencia de operaciones óptimas mediante un recorrido desde M[m][n] hasta M[0][0], comparando los costos para determinar qué operación llevó al estado actual.

E. Informe de complejidades del problema de la terminal inteligente

*1) **Solución de fuerza bruta de la terminal inteligente**:* La solución implementada mediante fuerza bruta explora todas las combinaciones de operaciones posibles. Cada combinación de operaciones es aplicada a la cadena inicial y se verifica si dicha secuencia de operaciones es la adecuada.

Este enfoque es el más costoso ya que implica probar todas las combinaciones posibles y en el peor caso la correcta podrá estar de última.

Para la resolución del problema de la terminal siguiendo este enfoque, nos centramos en primero generar todas las combinaciones posibles hasta un limite máximo de cantidad de operaciones, llamemos le a este limite $\boldsymbol{d}$ (el cual también, si lo vemos con un enfoque de búsqueda, hace referencia a la profundidad de búsqueda). Ademas tengamos en cuenta la cantidad de operaciones que vamos a tener, las cuales serán 5 operaciones, a estas operaciones asignemos le la variable $\boldsymbol{n}$.

Ahora con esto, podemos identificar las combinaciones de operaciones, que están dadas por la formula $\boldsymbol{n^d}$. Por lo tanto, la cantidad de secuencias de operaciones crece exponencialmente con la profundidad $\boldsymbol{d}$. Para cada incremento en la profundidad, el número de secuencias de operaciones aumenta exponencialmente.

Por ejemplo:

- Si $\boldsymbol{d = 1}$, hay 5 posibles secuencias.
- Si $\boldsymbol{d = 2}$, hay 5^2 = 25 posibles secuencias.
- Si $\boldsymbol{d = 3}$, hay 5^3 = 125 posibles secuencias.

Y así sucesivamente hasta ***d = maxima profundidad***.

También debemos considerar el costo de aplicar las operaciones, el cual es un costo constante $\boldsymbol{O(1)}$ ya que cada una de estas operaciones afecta la cadena de manera local. Considerando lo anterior Supongamos que el largo de la cadena inicial es $\boldsymbol{m}$ y el largo de la cadena final es $\boldsymbol{k}$. En el peor de los casos, aplicar una secuencia completa de $\boldsymbol{d}$ operaciones tiene un costo de $\boldsymbol{O(m + k)}$(sumando el costo de todas las operaciones en esa secuencia).

Por lo tanto, el costo total de tiempo para cada combinación de operaciones es proporcional a $\boldsymbol{O(d * (m + k))}$, ya que se realizan d operaciones para modificar la cadena.

Sumando todo lo anterior, la complejidad temporal total del enfoque de fuerza bruta es:

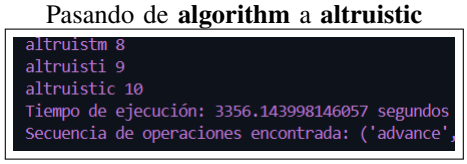
$$\boldsymbol{O(n^d * d * (m + k))}$$

Desglosando esta complejidad obtenemos que

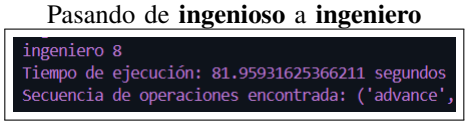
- n^d es el número de combinaciones de operaciones
- d es el número de operaciones en cada secuencia
- $m + k$ es el tamaño de las cadenas implicadas

A continuación se evidencias algunas capturas con ejemplos de ejecución del algoritmo, resaltando el tiempo que tomo encontrar la solución

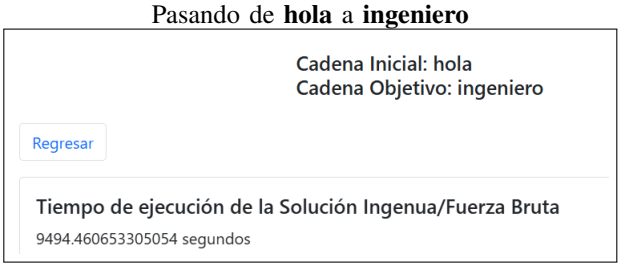
Pasando de **algorithm** a **altruistic**



Pasando de **ingenioso** a **ingeniero**



Pasando de **hola** a **ingeniero**



*2) **Solución Dinámica de la terminal inteligente**:* El problema de la terminal inteligente busca determinar el costo mínimo de transformar una cadena **X** en otra cadena **Y** utilizando un conjunto de operaciones con costos específicos. Estas operaciones incluyen avanzar, insertar, eliminar, reemplazar y kill (eliminación completa).

El objetivo principal es calcular una solución óptima que minimice el costo total y determinar la secuencia exacta de operaciones necesarias.

- **1. construcción de la matriz de costos (M):**
  - En esta etapa se llena una matriz de dimensiones m*n, donde m es la longitud de X y n es la longitud de Y.
  - Cada celda de la matriz se calcula en función de los valores previamente almacenados, lo que garantiza un enfoque dinámico y evita redundancias en los cálculos.
- **2. Recosntrucción de la solución óptima:**
  - A partir de la matriz de costos M, se realiza un recorrido hacia atrás para identificar la secuencia óptima de operaciones.

Complejidad temporal: la complejidad temporal del algoritmo es **O(n*m)**. Esto se debe a que para llenar la matriz M, el algoritmo realiza un ciclo anidado sobre **i** ∈ **[0,m]** y **j** ∈ **[0,n]**. Dentro de cada iteración, las operaciones realizadas (mínimos, comparaciones) son de tiempo constante, lo que resulta en O(m*n).

Complejidad espacial: la complejidad espacial del algoritmo es **O(n*m)**. Esto se debe a que se utiliza una matriz de tamaño m*n para almacenar los costos acumulados. Cada celda de la matriz almacena un valor entero, lo que resulta en un espacio total de O(m*n). Espacialmente, los costos adicionales (variables, almacenamiento de operaciones) son insignificantes en comparación.

Analisis de casos de prueba:

- **Mejor caso:**
  - **Situación:** Las cadenas X y Y son idénticas.
  - **Costo:** Solo se realizan operaciones de avance.
  - **Complejidad:** Aunque conceptualmente el costo total es menor, el algoritmo sigue recorriendo toda la matriz, lo que mantiene la complejidad en O(m*n)
- **Peor caso:**
  - **Situación:** Las cadenas X y Y son completamente diferentes.
  - **Costo:** Cada operación posible (insertar, eliminar, reemplazar, etc.) se evalúa, y la matriz se llena completamente.
  - **Complejidad:** La complejidad sigue siendo O(m*n), ya que el algoritmo debe recorrer toda la matriz para calcular los costos.

*3) **Solución Voraz de la terminal inteligente**:*

F. Soporte de complejidad computacional

*1) **Solución de fuerza bruta de la terminal inteligente**:* La solución por fuerza bruta o ingenua es la más demorada, siendo su complejidad $O(n^d * d * (m + k))$ ya discutida en el anterior apartado. Esta implementación se soporta en la idea de búsqueda por profundidad, asignando un límite d a esta búsqueda, por tanto los tiempos de ejecución dependen en gran medida del límite asignado, ya que en el peor de los casos la secuencia correcta de combinaciones puede estar en la profundidad límite lo que genera el mayor tiempo de ejecución.

Entre más caracteres tengan las cadenas, más operaciones deben ser aplicadas para encontrar la solución, lo que involucra combinaciones de operaciones más grandes y en general un tiempo de ejecución mucho más grande. Siguiendo esa idea, **decidimos tomar cadenas de una longitud pequeña**, en la que se evidencie el cambio sin que tome mucho tiempo obtener los datos promedio de ejecución, **ya que para cadenas de longitud media o grande con al menos 50 repeticiones (con el fin de promediar) hasta una profundidad 20**, llegan a demorarse en **promedio más de 12 horas**, lo cual nos resultó en un problema para tomar estos datos correctamente.

Para estas medidas decidimos tomar desde la longitud de la **cadena inicial + 1 hasta la profundidad 20** esto con el fin de obtener una respuesta, ya que si dejamos una profundidad muy corta (menor a la cantidad de caracteres) no existirá margen para que la secuencia de operaciones pueda resolver el problema.

En la siguiente tabla vemos los tiempos para distintas profundidades de transformar la cadena **hola** en la cadena **hali**

Profundidad (d)	Tiempo de ejecución promedio (s)
5	0.069946760002058
6	0.06405548200011253
7	0.061869512000121175
8	0.061996816000901164
9	0.06429767200024798
10	0.06056333400076255
11	0.062394928000867364
12	0.06560532600153238
13	0.06023775800131261
14	0.0599320979998447
15	0.06116698800120503
16	0.059828953999094665
17	0.061609809999354184
18	0.06138069000095129
19	0.06272568400017917
20	0.06057886200025678

Como podemos ver, en este caso los tiempos son muy similares, siendo en promedio para todas las profundidades de **0.06s**, lo que significa que la secuencia de combinaciones de operaciones se encuentra de forma fácil, evitando de esta forma el peor caso, el cual sería encontrarla en las últimas secuencias.

En la siguiente tabla vemos los tiempos para distintas profundidades de transformar la cadena **hello** en la cadena **hell**

Profundidad (d)	Tiempo de ejecución promedio (s)
6	0.29513427800033243
7	0.28443249200005083
8	0.283537137999665
9	0.2853603720013052
10	0.27768911599880086
11	0.2873572580004111
12	0.28049664200050756
13	0.2795284239994362
14	0.27905826000031086
15	0.281818458000198
16	0.3484723600000143
17	0.5985765219992026
18	0.46111258800141514
19	0.294363985999953
20	0.2728795140003786

En este problema, aumentan los tiempos de ejecución, pensaríamos que inicialmente es sencillo encontrar una solución solo borrando el ultimo carácter, pero la naturaleza de la solución por fuerza bruta nos demuestra que no importa cual es la mejor solución, solo se decanta por la primera que encuentra.

En la siguiente tabla vemos los tiempos para distintas profundidades de transformar la cadena **gomelo** en la cadena **gomita**

Profundidad (d)	Tiempo de ejecución promedio (s)
7	2.7486599879999996
8	2.2181410059999997
9	1.881568568
10	2.804815637999999
11	2.1894951219999985
12	1.8541460920000008
13	1.8414377100000001
14	1.9194692860000009
15	1.9021380980000004
16	2.876983774
17	2.257403501999997
18	1.8678160259999959
19	1.8047060360000022
20	1.7986948740000026

En este problema podemos observar diferencias más significativas entre algunas profundidades de búsqueda, acá es donde podemos evidenciar que la profundidad afecta el rendimiento, sobretodo en casos en los que existen muchas soluciones, si el algoritmo se inclina por una secuencia que no es la correcta se pierde tiempo, pero si existen distintas secuencias correctas, entre más profundidad, mayor posibilidades de encontrar una que funcione.

En la siguiente tabla veremos tiempos para distintas profundidades de transformar la cadena **patria** a **patriota**

Profundidad (d)	Tiempo de ejecución promedio (s)
7	12.270462038000005
8	10.886457358000007
9	10.871175563999987
10	10.951866476000014
11	10.839189434
12	10.87935591200001
13	10.899951377999987
14	10.907528085999983
15	10.952159657999982
16	11.946722919999992
17	13.330963043999981
18	14.141401055999996
19	15.014239315999985
20	10.91898668400001

Como podemos ver, esta ha sido hasta ahora la que más tiempo ha tomado y ademas cuenta con variaciones de tiempos promedios más alta de lo común, ya que sucede lo anteriormente mencionado, dependiendo de la profundidad de búsqueda puede encontrar soluciones más rápidamente o no.

*2) **Solución Dinámica Terminal Inteligente**:* En esta parte lo que hemos hecho es sacar tiempos ejecutando un mismo problema de combercion de cadenas y luego le sacamos los datos de tiempo de ejecución con tal de ver si el paradigma de la programación dinamica es mucho mas efectiva, en este caso tomamos la conversión de la cadena **Electroencefalografista**

a la cadena **Esternocleidomastoideo**.

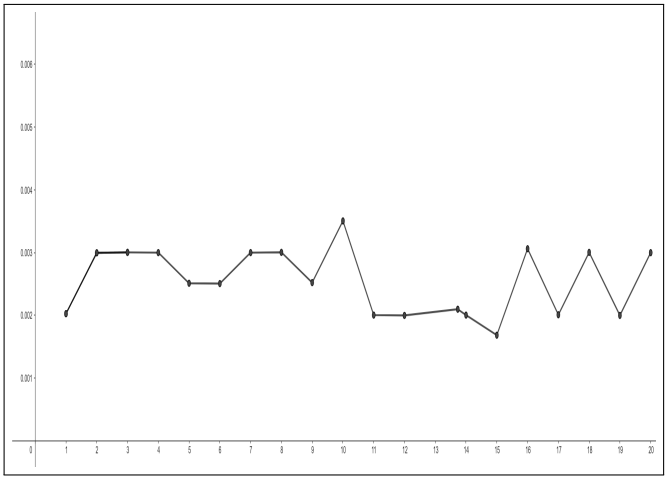
Lo primero que podemos destacar es que a comparacion de la solucion por fuerza bruta, esta no esta siendo limitada por la cantidad de letras en la cadena, como si pasa en la de fuerza bruta, llegando a tener tiempos extremos de demora comparados con los que se presentan a continuacion:

- **Tabla:**

iteracion	tiempo
1	0.002027273178100586
2	0.0029973983764648438
3	0.0030045509338378906
4	0.0030007362365722656
5	0.002512216567993164
6	0.002507448196411133
7	0.0030007362365722656
8	0.0030050277709960938
9	0.002523183822631836
10	0.0035059452056884766
11	0.0020051002502441406
12	0.0019998550415039062
13	0.0019998550415039062
14	0.0020051002502441406
15	0.0016849040985107422
16	0.0030624866485595703
17	0.0020101070404052734
18	0.0030045509338378906
19	0.002000093460083008
20	0.0029997825622558594

Tiempo de ejecución en segundos

- **Grafica:**



lo que podemos ver es que la solución dinámica es muy rápida, esto teniendo en cuenta de que estamos trabajando con una cadena que tiene 23 caracteres, y en comparación de la solución por fuerza bruta que se demoraría mucho tiempo en resolver el problema, el resultado promedio de tiempo en este caso es 0.0025478327751159668 segundos. Ahora probemos con una cadena de menos caracteres haber si estos tiempo mejoran en relación al tamaño de las cadenas lo que llevaría a hacer mas pasos.

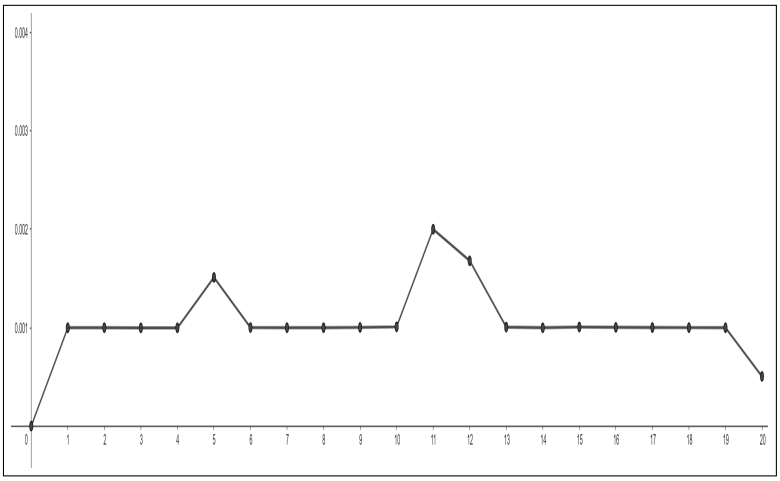
En este caso usaremos la secuencia de transformar la cadena **ingenioso** a la cadena **ingeniero**. Lo que podríamos esperar que pase es que al ser menos caracteres los tiempos de ejecución sean menores, primero ejecutemos el algoritmo la misma cantidad de veces que en el caso anterior para tener una comparación justa.

- **Tabla:**

Iteracion	Tiempo
1	0.0009999275207519531
2	0.0010001659393310547
3	0.0009987354278564453
4	0.0009989738464355469
5	0.0015110969543457031
6	0.0010018348693847656
7	0.0010004043579101562
8	0.0009999275207519531
9	0.0010030269622802734
10	0.001008749008178711
11	0.0020008087158203125
12	0.0016798973083496094
13	0.0010061264038085938
14	0.0009996891021728516
15	0.001007080078125
16	0.0010037422180175781
17	0.001001596450805664
18	0.001001119613647461
19	0.0010001659393310547
20	0.0005052089691162109

Tiempo de ejecución en segundos

- **Grafica:**



En este caso podemos ver que los tiempos de ejecución son mucho menores, lo que nos lleva a que realmente la solución

depende de varios factores, como lo puede ser la cantidad de caracteres en las cadenas, o la cantidad de caracteres que son iguales en ambas cadenas, en este ejercicio el tiempo promedio fue de **0,00108641386032104** un numero mucho menor al anterior.

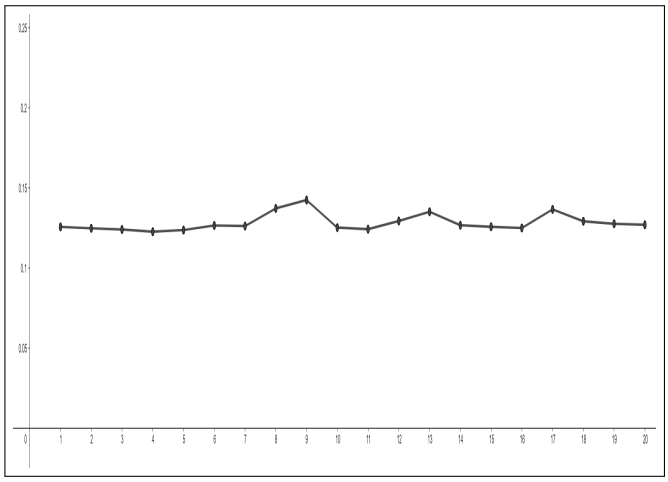
Para finalizar haremos un ultimo ejemplo en el que trataremos de estresar al máximo el algoritmo, buscaremos que nos de tiempos mas elevados enviándoles cadenas muy largas con mas de 200 caracteres, sacamos de un lorem ipsum varias secuencias de texto y las mandaremos para ver como se comporta el algoritmo, las cadenas son las siguientes:

```
X =
"Loremipsumdolorsitametconsecteturadipiscingelitseddoeiusmodtemporaliquipexeacommod
oconsequatLoremipsumdolorsitametconsecteturadipiscingelitseddoeiusmodtemporaliquipe
xeacommodoconsequatLoremipsumdolorsitametconsecteturadipiscingelitseddoeiusmodtempo
raliquipexeacommodoconsequatLoremipsumdolorsitametconsecteturadipiscingelitseddoeiu
smodtemporaliquipexeacommodoconsequatLoremipsumdolorsitametconsecteturadipiscingeli
tseddoeiusmodtemporaliquipexeacommodoconsequatLoremipsumdolorsitametconsecteturadip
iscingelitseddoeiusmodtemporaliquipexeacommodoconsequat"
Y =
"incididuntutlaboreetdoloremagnaaliquaUtenimadminimveniamquisnostraliquipexeacommod
oconsequatincididuntutlaboreetdoloremagnaaliquaUtenimadminimveniamquisnostraliquipe
xeacommodoconsequatincididuntutlaboreetdoloremagnaaliquaUtenimadminimveniamquisnost
raliquipexeacommodoconsequatincididuntutlaboreetdoloremagnaaliquaUtenimadminimvenia
mquisnostraliquipexeacommodoconsequatincididuntutlaboreetdoloremagnaaliquaUtenimadm
inimveniamquisnostraliquipexeacommodoconsequatincididuntutlaboreetdoloremagnaaliqua
Utenimadminimveniamquisnostraliquipexeacommodoconsequat"
```

- **Tabla:**

Iteracion	Tiempo
1	0,125660419464111000
2	0,124787092208862000
3	0,124055385589599000
4	0,122645139694213000
5	0,123692750930786000
6	0,126544952392578000
7	0,126192331314086000
8	0,137213706970214000
9	0,142436027526855000
10	0,125214099884033000
11	0,124217510223388000
12	0,129369735717773000
13	0,135107517242431000
14	0,126693487167358000
15	0,125752687454223000
16	0,124997377395629000
17	0,136578559875488000
18	0,129174232482910000
19	0,127621173858642000
20	0,126975059509277000

- **Grafica:**



Efectivamente lo que hicimos fue estresar al algoritmo, y como podemos ver los tiempos de ejecución son mucho mayores, pero en comparación con las primeras ejecución mas si embargo los tiempos siguen siendo muy buenos lo que nos puede decir que la programación dinámica es muy efectiva para resolver este tipo de problemas, es este caso el tiempo promedio fue de **0,128246462345123000** segundos.

3) ***Solución Voraz Terminal Inteligente****:* El algoritmo voraz para transformar una cadena en otra funciona tomando decisiones paso a paso, seleccionando en cada iteración la operación que parece más prometedora para acercar la cadena actual a la cadena objetivo. Esto se hace evaluando cada una de las opciones como avanzar, reemplazar, insertar, eliminar o ”matar” caracteres, y elegirá aquella que maximice las coincidencias con la cadena objetivo o minimice la longitud excedente.

Cada decisión se basa únicamente en el estado actual de la cadena, es decir no se considera el efecto que tendrá esa decisión en pasos futuros. Este enfoque busca resolver el problema optimizando localmente en cada paso, con la esperanza de que las decisiones locales conduzcan a una solución global. Sin embargo, esto no garantiza que siempre se encuentre la solución óptima global.

La complejidad computacional de este algoritmo está determinada principalmente por la función `evaluar_opciones`, que se define de la siguiente manera:

```
def evaluar_opciones(opciones, cadenaObjetivo):
    evaluaciones = []

    for opcion in opciones:
        cadena, pos, operacion = opcion
        coincidencias = guia(cadena, cadenaObjetivo)
        evaluaciones.append((cadena, pos, operacion,
             coincidencias))

    return evaluaciones
```

Esta función recibe una lista de tuplas, donde cada tupla contiene: la cadena modificada, la última posición procesada, y la operación realizada para obtener dicha modificación. La función evalúa cada una de estas opciones y agrega, como un cuarto elemento de la tupla, el número de caracteres que coinciden en la misma posición con la cadena objetivo.

El costo computacional de esta función es $O(k \cdot n)$, donde k es el número de opciones evaluadas y n es el costo de invocar la función `guía`, que se encarga de contar los caracteres coincidentes.

Con esto en mente, podemos analizar la función `transformar_cadena_voraz`, que es la encargada de llamar a la función `evaluar_opciones` y determinar la complejidad final de nuestro algoritmo. El código de esta función, con partes omitidas y resumidas para mayor claridad, es el siguiente:

```
def transformar_cadena_voraz(cadenaOriginal, cadenaObjetivo
    ):
    #
        #Inicializa variables
    #

    while True:
        #
            #Planteamiento del caso base
        #
        opciones = []

        # Opcion de avanzar si el caracter actual coincide
        if posicion < len(cadenaOriginal) and posicion <
             len(cadenaObjetivo) and cadenaOriginal[posicion
             ] == cadenaObjetivo[posicion]:
            nueva_posicion = advance(posicion)
            opciones.append((cadenaOriginal[:],
                 nueva_posicion, "advance"))

        #
            #Condicionales para determinar la operacion a
                 realizar
            #(reemplazar,insertar,matar,eliminar)
        #

        # Evaluar todas las opciones
        evaluaciones = evaluar_opciones(opciones,
             cadenaObjetivo)
```

Al observar el código, notamos que existe un ciclo `while` que se ejecutará tantas veces como sea necesario para transformar completamente la `cadenaOriginal` en la `cadenaObjetivo`. Denotamos este número de iteraciones con n, que representa el tamaño máximo entre ambas cadenas.

Dentro del ciclo, la función evalúa qué operaciones realizar (como insertar, eliminar, reemplazar o matar), dependiendo de si las condiciones específicas se cumplen. Cada operación válida se añade a la variable `opciones`. Una vez que se generan estas opciones, la función llama a `evaluar_opciones`, que recibe estas alternativas y ejecuta su lógica previamente explicada.

Dado que la función `evaluar_opciones` tiene un costo de $O(k \cdot n)$, y el ciclo `while` se ejecuta $O(n)$ veces en el peor caso, la complejidad computacional total del algoritmo es:

$$O(n^2 \cdot k)$$

Mediciones de tiempo:

A continuación se presenta una tabla con las mediciones de tiempo para cinco tipos diferentes de cadenas. Cada medición corresponde al promedio de 50 ejecuciones para cada uno de los ejemplos.

Gráficos de la complejidad teórica y experimental

CADENA ORIGINAL	CADENA OBJETIVO	TIEMPO
hola	pato	0.000016s
electricidad	electrolitros	0.000049s
planificacionestrategica	implementacionpractica	0.000126s
transformacionpoderosa	deformacioncontrolable	0.000122s
Otorrinolaringologicalisimamente	Electroencefalografista	0.000129s

TABLE I
MEDICIONES DE TIEMPO PARA DIFERENTES CADENAS

Para graficar la complejidad teórica, se utilizaron los mismos 5 ejemplos mencionados anteriormente. La cota asintótica se tomó como $O(n^2)$, dado que, en el peor de los casos, el valor de k es 3, lo cual se considera una constante que no varía significativamente en nuestra cota. Los resultados obtenidos utilizando estos ejemplos se muestran en la siguiente tabla:

CADENA ORIGINAL	CADENA OBJETIVO	Tamaño	f(n)
hola	pato	4	16
Electricidad	electrolitros	12	144
planificacionestrategica	implementacionpractica	24	576
transformacionpoderosamente	deformacioncontrolable	26	676
Otorrinolaringologicalisimamente	Electroencefalografista	32	1024

TABLE II
RESULTADOS DE LA COMPLEJIDAD TEÓRICA PARA DIFERENTES CADENAS

La gráfica que representa esa cota asintótica es la siguiente:

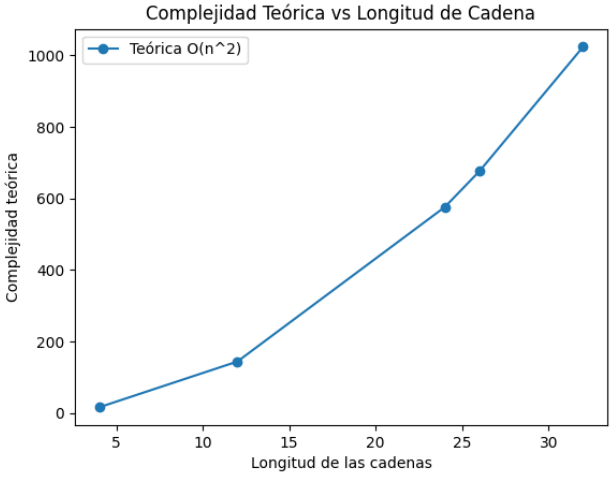


Fig. 1. Gráfico de la complejidad teórica

A continuación, se presentará una comparación entre dos gráficas: una que muestra el tiempo experimental y otra que representa la complejidad teórica. La línea azul ilustrará el crecimiento del tiempo de ejecución en función del tamaño de la cadena, mientras que la línea punteada reflejará la complejidad teórica, que describe el comportamiento esperado del crecimiento de nuestro algoritmo.

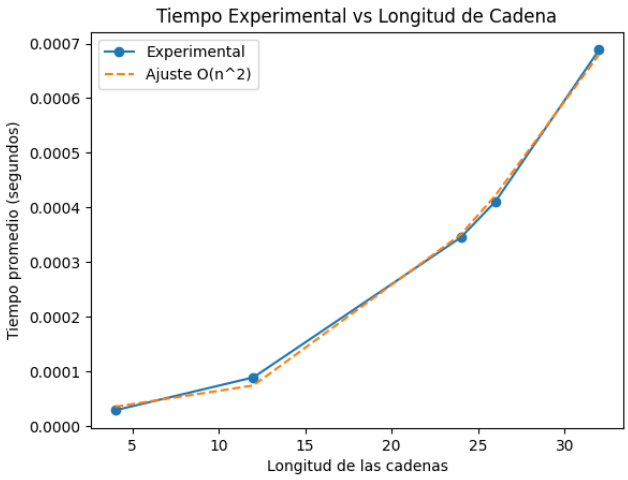


Fig. 2. Gráfico experimental vs teórica

A simple vista, ambos crecimientos son prácticamente idénticos. Sin embargo, es importante señalar que esto no siempre será así, ya que la ejecución experimental puede verse influenciada por factores como la calidad de los componentes del ordenador o la precisión de las mediciones de tiempo. Debido a estos factores, ya sean internos en el proceso de medición o relacionados con el hardware, la gráfica puede mostrar un comportamiento diferente, como se muestra a continuación.

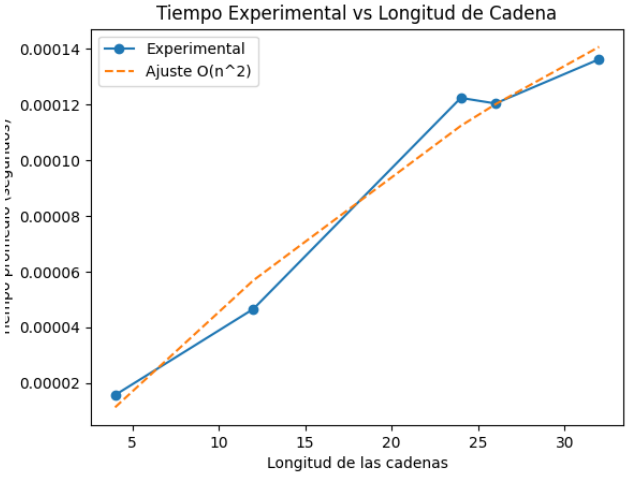


Fig. 3. Ejemplo

II. El problema de la Subasta Pública

A. *Entender El problema*

En esta parte se nos pide que para ver si hemos entendido el problema, se nos pide que hagamos dos asignaciones de las acciones para A = 1000, B = 100, n = 2, la oferta $< \mathbf{500, 100, 600} >$, la oferta$< \mathbf{450, 400, 800} >$ y la oferta del gobierno $< \mathbf{100, 0, 1000} >$. Indique el valor vr para la solución.

- **Primera asignación:**
  - Oferente 1: Se asignan 600 acciones (máximo permitido).
  - Oferente 2: Se asignan 400 acciones (mínimo permitido).
  - Gobierno : Se asignan 0 acciones (no es necesario que compre, puesto que ya tenemos las 1000 acciones de A).
  - **calculo del ingreso (vr):**

```
vr=(600*500)+(400*450)+(0*100)
  =300.000+ 180.000 + 0 =
  480.000
```

- **Segunda asignación:**
  - Oferente 1: Se asignan 100 acciones (mínimo permitido).
  - Oferente 2: Se asignan 800 acciones (máximo permitido).
  - Gobierno: Se asignan 100 acciones (para completar A=1000).
  - **calculo del ingreso (vr):**

```
vr=(100*500)+(800*450)+(100*100)
  =50.000+360.000+10.000 =
  420.000
```

En este ejemplo podemos ver dos diferentes asignaciones para la compra de las acciones, en este caso la primera asignación la podemos tomar como la solución optima, ya que maximizamos el valor recibido.

B. *Una primera aproximación*

Una primera aproximación al problema de la subasta pública podría ser considerar el ejercicio puesto en el enunciado del Proyecto en el que A = 1000, B = 100, n = 4, $< \mathbf{500, 400, 600} >$, $< \mathbf{450, 100, 400} >$, $< \mathbf{400, 100, 400} >$, $< \mathbf{200, 50, 200} >$, la oferta del gobierno $< \mathbf{100, 0, 1000} >$, para estas tenemos 2 asignaciones, una optima y otra valida pero no optima.

- **Asignación optima:**
  - Oferente 1: Se asignan 600 acciones (máximo permitido).
  - Oferente 2: Se asignan 400 acciones (mínimo permitido).
  - Oferente 3: Se asignan 0 acciones (no es necesario que compre, puesto que ya tenemos las 1000 acciones de A).
  - Oferente 4: Se asignan 0 acciones (no es necesario que compre, puesto que ya tenemos las 1000 acciones de A).
  - Gobierno: Se asignan 0 acciones (no es necesario que compre, puesto que ya tenemos las 1000 acciones de A).
  - **calculo del ingreso (vr):**

```
vr=(600*500)+(400*450)+(0*400)+(0*200)
    +(0*100)=300.000+180.000+0+0+0 =
    480.000
```

- **Asignación valida pero no optima:**
  - Oferente 1: Se asignan 400 acciones (minimo permitido).
  - Oferente 2: Se asignan 200 acciones
  - Oferente 3: Se asignan 300 acciones
  - Oferente 4: Se asignan 50 acciones
  - Gobierno: Se asignan 50 acciones (para completar A=1000).
  - **calculo del ingreso (vr):**

```
vr=(400*500)+(200*450)+(300*400)
    +(50*200)+(50*100)
    =200.000+90.000+120.000+10.000+5.000
     = 425.000
```

C. *Caracterizar la estructura de la solución óptima*

La estructura de la solución óptima en el algoritmo de subasta dinámica está basada en el principio de programación dinámica, lo que implica que la solución óptima se construye a partir de soluciones óptimas de subproblemas más pequeños. En este caso, el problema se descompone en subproblemas que buscan maximizar el ingreso total dado un número de oferentes y un conjunto de acciones disponibles, mientras se cumplen las restricciones de cada oferente.

Subestructura óptima:
La solución óptima del problema se construye a partir de soluciones óptimas parciales. En cada paso, el algoritmo evalúa el máximo ingreso posible que se puede obtener con un número específico de acciones asignadas a un conjunto determinado de oferentes. La clave aquí es que se selecciona la mejor asignación para cada oferente, considerando las restricciones de cada uno y la maximización del ingreso total. Las decisiones previas influyen en el comportamiento de las futuras asignaciones, lo que permite construir una solución que satisface todas las restricciones y maximiza el ingreso.

Matriz de soluciones:
El algoritmo utiliza una tabla bidimensional (matriz) `subasta`, donde cada entrada `subasta[i][j]` representa el máximo ingreso que se puede obtener considerando los primeros i oferentes y exactamente j acciones asignadas. Esta matriz permite que cada subproblema dependa solo de subproblemas más pequeños, y cada solución parcial se utiliza para construir la solución global.

Decisiones de asignación:
La solución óptima se basa en decisiones locales que maximizan el ingreso en cada paso, seleccionando la cantidad de acciones que se deben asignar a cada oferente dentro de sus restricciones. En el caso de cada oferente, el algoritmo evalúa todas las posibles cantidades de acciones que puede recibir, desde el mínimo requerido hasta el máximo permitido, y elige la opción que maximiza el ingreso total sin exceder el número de acciones disponibles.

Reconstrucción de la solución:
Una vez que el algoritmo ha llenado la matriz de soluciones, se puede reconstruir la asignación óptima de acciones para cada oferente. Comenzando desde la última fila y la última columna de la matriz (donde se encuentra la solución óptima global), se rastrean las decisiones tomadas en cada paso para determinar cuántas acciones se asignan a cada oferente.

Garantía de optimalidad:
La solución es garantizada como óptima porque, para cada número de acciones asignadas, el algoritmo evalúa todas las posibles asignaciones que se pueden hacer en función de los oferentes y sus restricciones. Como todas las combinaciones posibles se exploran, se garantiza que se selecciona la mejor opción en cada paso para maximizar el ingreso total.

D. *Definir recursivamente el valor de la solución óptima*

El algoritmo de la subasta pública puede definirse recursivamente utilizando la siguiente fórmula. Sea $subasta[i][j]$ el valor óptimo (el ingreso máximo) cuando estamos considerando los primeros i oferentes y tenemos j acciones disponibles. La solución recursiva está dada por las siguientes reglas:

Casos base

- $subasta[0][0] = 0$: Si no hay oferentes y no hay acciones, el ingreso máximo es 0.
- $subasta[i][0] = 0$: Si hay oferentes pero no hay acciones, el ingreso máximo también es 0, ya que no se puede realizar ninguna transacción.

Relación de recurrencia

Para $i > 0$ y $j > 0$, la matriz se actualiza siguiendo estas reglas:

1. **No asignar acciones al oferente** i:

$$subasta[i][j] = subasta[i-1][j]$$

Esto significa que si no asignamos acciones al oferente i, entonces el valor óptimo será igual al valor óptimo considerando solo los primeros $i-1$ oferentes con j acciones disponibles.

2. **Asignar** n **acciones al oferente** i: Si asignamos n acciones al oferente i, donde $m_i \leq n \leq M_i$ (con m_i y M_i siendo la cantidad mínima y máxima de acciones que el oferente i está dispuesto a comprar), entonces el valor óptimo será el valor máximo de la suma del valor de las transacciones previas más el costo de asignar n acciones al oferente i:

$$subasta[i][j+n] = \max(subasta[i][j+n], subasta[i-1][j]+P_i \cdot n)$$

donde P_i es el precio por acción del oferente i, y n es el número de acciones asignadas al oferente i.

El valor máximo que se puede obtener es el valor contenido en la celda $subasta[N][A]$, donde N es el número total de oferentes y A es el número total de acciones.

E. *Calcular el valor de una solución óptima*

Parámetros de entrada

- $A = 5$ (número de acciones disponibles)
- $B = 1$ (precio mínimo por acción ofertado por el gobierno)

- Oferentes:
  - Oferente 1: $P = 5$, $m = 1$, $M = 4$
  - Oferente 2: $P = 3$, $m = 2$, $M = 5$

Paso 1: Inicialización de la matriz

Creamos una matriz de tamaño $(N+1) \times (A+1)$, donde N es el número de oferentes (incluyendo al gobierno) y A es el número de acciones. Inicializamos todas las celdas con $-\infty$, excepto la celda $[0][0] = 0$, que representa el caso base (sin oferentes ni acciones).

Matriz inicial:

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

Paso 2: Procesar al oferente 1 ($P = 5$, $m = 1$, $M = 4$)

Evaluamos:

- **No asignar acciones**: copiamos el valor de la fila anterior.
- **Asignar n acciones ($m \leq n \leq M$)**: calculamos el ingreso adicional y actualizamos si mejora el valor actual.

Matriz después de procesar al oferente 1:

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 0 & 5 & 10 & 15 & 20 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

Paso 3: Procesar al oferente 2 ($P = 3$, $m = 2$, $M = 5$)

Evaluamos:

- **No asignar acciones**: copiamos el valor de la fila anterior.
- **Asignar n acciones ($m \leq n \leq M$)**: calculamos el ingreso adicional y actualizamos si mejora el valor actual.

Matriz después de procesar al oferente 2:

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 0 & 5 & 10 & 15 & 20 & -\infty \\ 0 & 5 & 10 & 15 & 20 & 21 \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

Paso 4: Procesar al gobierno ($P = 1$, $m = 0$, $M = 5$)

El gobierno puede comprar cualquier cantidad de acciones restantes sin incrementar el ingreso máximo.

Matriz después de procesar al gobierno:

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 0 & 5 & 10 & 15 & 20 & -\infty \\ 0 & 5 & 10 & 15 & 20 & 21 \\ 0 & 5 & 10 & 15 & 20 & 21 \end{bmatrix}$$

Paso 5: Recuperar la solución

El ingreso máximo se encuentra en la celda $[N][A] = 21$. Para encontrar la asignación:

- Rastreamos desde la celda $[N][A]$ hacia atrás.
- Identificamos cuántas acciones se asignaron a cada oferente.

Resultado final

- **Ingreso máximo**: 21
- **Asignación**: $[3, 2, 0]$ (Oferente 1: 3 acciones, Oferente 2: 2 acciones, Gobierno: 0 acciones)

F. Construir una solución óptima

Una vez que hemos calculado la matriz $subasta[i][j]$ que contiene el ingreso máximo posible para los primeros i oferentes con j acciones disponibles, podemos construir la solución óptima siguiendo los siguientes pasos:

1. Identificación del valor óptimo

El valor óptimo de la subasta, es decir, el ingreso máximo que se puede obtener, se encuentra en la celda $subasta[N][A]$, donde N es el número de oferentes (incluyendo al gobierno) y A es el número total de acciones disponibles.

$$\text{Ingreso máximo} = subasta[N][A]$$

2. Rastreo de la asignación de acciones

Para encontrar cómo se distribuyeron las acciones entre los oferentes, debemos rastrear el valor de $subasta[i][j]$ desde $subasta[N][A]$ hacia atrás. El objetivo es determinar cuántas acciones fueron asignadas a cada oferente. Los pasos son los siguientes:

- Comenzamos en la celda $[N][A]$.
- Si $subasta[i][j] = subasta[i-1][j]$, significa que no se asignaron acciones al oferente i, por lo que pasamos a la fila anterior $i - 1$.
- Si $subasta[i][j] \neq subasta[i-1][j]$, entonces se asignaron n acciones al oferente i. Debemos encontrar el valor n tal que:

  $$subasta[i][j] = subasta[i-1][j-n] + P_i \cdot n$$

  donde P_i es el precio por acción del oferente i y n es el número de acciones asignadas. Restamos n de j para obtener la cantidad restante de acciones.
- Registramos la cantidad n de acciones asignadas al oferente i y luego actualizamos $j = j - n$.
- Repetimos el proceso hasta llegar a la celda $[1][0]$.

3. Resultado final

La distribución de las acciones entre los oferentes estará representada por una lista de la forma:

$$\text{Asignación} = [n_1, n_2, \ldots, n_N]$$

donde n_i es el número de acciones asignadas al oferente i.

El ingreso máximo es el valor en la celda $subasta[N][A]$, y la asignación óptima es la lista obtenida en el paso 2.

G. *Informe de complejidades del problema de la subasta pública*

*1) **Solución de fuerza bruta problema de la subasta pública**:* En el caso de la subasta publica tenemos que es una propuesta que nos da la solución optima en la mayoría sino es en todos los casos, pero tiene un problema muy grande y es que la complejidad computacional es muy alta, ya que se tiene que probar todas las posibles combinaciones de asignaciones de acciones para los oferentes, y esto se hace de manera recursiva, lo que nos lleva a que la complejidad computacional sea muy alta.

una muestra de ello podría ser el siguiente ejemplo:

- Si tienes n oferentes, cada uno con un rango de asignaciones de R_i = M_i-m_i+1, el número total de combinaciones posibles es:

$$\prod_{i=1}^{n} R_i$$

esto quiere decir si los oferentes tienen rangos de 1001 cada uno, y hay 3 oferentes, el número total de combinaciones sería 1001^3=1,003,003,001 combinaciones posibles. lo que podría llevar a que el tiempo de ejecución sea muy alto, y en el peor de los casos el limite de memoria nos podría dar un error de ejecución.

- **Complejidad temporal:** Para cada oferente, hay un rango de valores desde m_i (mínimo) hasta M_i(máximo), lo que implica que el número total de combinaciones posibles es

$$\prod_{i=1}^{n}(M_i - m_i + 1)$$

Donde n+1 incluye el gobierno como oferente adicional- Para cada combinación se realizan suma de asignaciones (O(n)) y Calculo de ingreso (O(n)), esto da un costo de evaluación de **O(n)** por combinación. Por lo tanto la complejidad total temporal es:

$$O(\prod_{i=1}^{n}(M_i - m_i + 1) * n)$$

si todos los rangos son aproximadamente iguales, la complejidad se reduce a

$$O(R^{n+1} * n)$$

Donde R es el tamaño del rango de asignaciones.

- **Complejidad espacial:** En el algoritmo, las combinaciones se generan de manera incremental por lo que no almacenamos todas las combinaciones simultáneamente y el espacio ocupado por la generación de combinaciones es O(n), ya que necesitamos una combinación activa a la vez.
También tenemos el espacio para las variables de evaluación que en nuestro código son **mejor_ingreso**, **mejor_asignacion** y **acciones_gobierno** que ocupan un espacio constante O(1). En general, la complejidad total espacial del algoritmo es **O(n)**.

*2) **Solución Dinámica problema de la subasta pública**:* El algoritmo de subasta dinámica se utiliza para asignar un número limitado de acciones a varios oferentes de manera que se maximice el ingreso total, respetando las restricciones de cada oferente en cuanto al número mínimo y máximo de acciones que puede adquirir. El proceso se lleva a cabo paso a paso, evaluando en cada iteración las diferentes opciones de asignación de acciones a los oferentes, y seleccionando la cantidad de acciones que maximiza el ingreso total posible en función de las ofertas y restricciones de cada participante.

Se construye una tabla que almacena los ingresos máximos posibles para cada combinación de oferentes y acciones asignadas. Este enfoque garantiza que el algoritmo explore de manera eficiente todas las combinaciones posibles de asignación y genere una solución óptima.

El costo computacional $O(N \cdot A \cdot M)$ donde tenemos que: N = Cantidad de oferentes A = Cantidad total de acciones M = Cantidad maxima de acciones que un oferente puede comprar

Esto se determino por el siguiente fragmento de codigo:

```
    for i in range(1, N + 1):
        oferta = oferentes[i-1]
        P = oferta['P'] #Precio a pagar por cada accion
        m = oferta['m'] #Cantidad minima de acciones a
            comprar
        M = oferta['M'] #Cantidad maxima de acciones a
            comprar

        for j in range(A+1):
            #No asignar acciones al oferente
            if subasta[i-1][j] > subasta[i][j]:
                subasta[i][j] = subasta[i-1][j]


            #Asignar n acciones al oferente i
            # n puede ser desde el minimo hasta el maximo
                sin exceder A
            for n in range(m,min(M, A - j) + 1):
                if subasta[i-1][j] + P * n > subasta[i][j +
                    n]:
                    subasta[i][j + n] = subasta[i-1][j] + P
                        * n
```

1. Primer bucle (Sobre i)

El primer bucle recorre todos los oferentes, desde 1 hasta N. Por lo tanto, el número de iteraciones de este bucle es proporcional a N, lo que da una complejidad de:

$$O(N)$$

2. Segundo bucle (Sobre j)

El segundo bucle recorre todos los posibles valores de las acciones disponibles, desde 0 hasta A, lo que da $A + 1$ iteraciones. Por lo tanto, la complejidad de este bucle es:

$$O(A)$$

3. Tercer bucle (Sobre n)

El tercer bucle asigna un número de acciones n al oferente. Para cada valor de i y j, el rango de n está determinado por los valores m (cantidad mínima de acciones) y M (cantidad

máxima de acciones que puede comprar el oferente). En el peor caso, si M es similar a A, este bucle recorrerá hasta M iteraciones. Por lo tanto, la complejidad de este bucle es:

$$O(M)$$

Cálculo total del costo computacional

El costo total se obtiene multiplicando los costos de cada uno de los bucles anidados:

$$O(N \cdot A \cdot M)$$

Grafico de complejidad teorica A continuación, se presenta una tabla que muestra los datos utilizados para analizar el comportamiento de la gráfica teórica. Estos datos han sido calculados mediante la fórmula del costo computacional, a partir de un conjunto de valores que hemos proporcionado.

N	A	M	f(n)
4	20	20	1600
4	15	15	900
4	25	25	2500
4	18	18	1296
4	22	22	1936

TABLE III
DATOS UTILIZADOS LA GRÁFICA TEÓRICA.

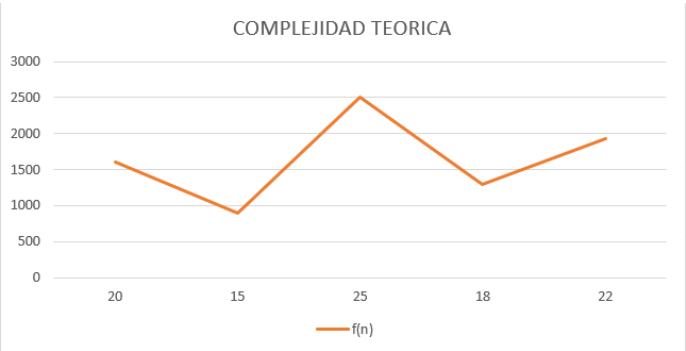


Fig. 4. Gráfico de la complejidad teórica

Grafico de complejidad experimental vs teorica A continuación, se presentan los datos de las mediciones de tiempo, en los cuales se calculó el promedio de 50 ejecuciones para cada uno de los 5 ejemplos. Estos datos servirán para analizar el comportamiento esperado de nuestra gráfica.

N	A	M	TIEMPO PROMEDIO
4	20	20	0.0001
4	15	15	0.00007
4	25	25	0.00021
4	18	18	0.00008
4	22	22	0.00012

TABLE IV
MEDICIONES DE TIEMPO

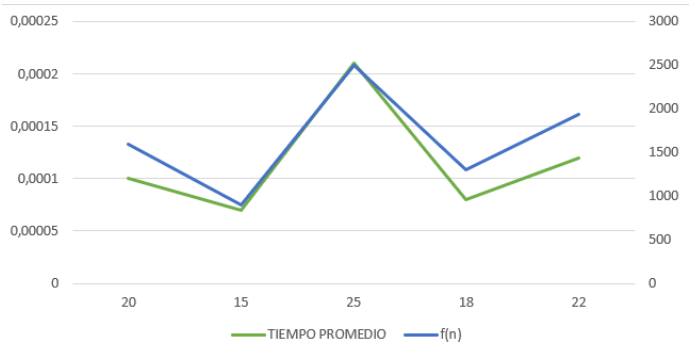


Fig. 5. Gráfico experimental

Como se puede observar, la línea verde representa la gráfica de la ejecución experimental, mientras que la línea azul corresponde a la complejidad teórica. Al comparar ambas, se nota que siguen un comportamiento similar, lo que nos permite inferir que el cálculo de la complejidad de nuestro algoritmo y el resultado esperado están en línea con lo predicho. Sin embargo, es importante destacar que el comportamiento de la ejecución experimental también dependerá de la capacidad del dispositivo en el que se ejecute el algoritmo.

*3) **Solución Voraz problema de la subasta pública**:* La solución implementada mediante un algoritmo voraz explora todas las ofertas, y selecciona entre ellas las que tienen un mejor precio, cumpliendo con las restricciones de $\boldsymbol{B}$ (precio mínimo), $\boldsymbol{m_i}$ (mínimo de acciones que se pueden comprar de acciones que se pueden comprar a dicho oferente) y $\boldsymbol{M_i}$ (máximo de acciones que se pueden comprar a dicho oferente). Por tanto, este algoritmo va a decidir en cada paso, cual es la mejor decisión que se puede tomar localmente, tratando de maximizar el valor recibido final $\boldsymbol{vr}$.

Inicialmente lo que se hace es ordenar las ofertas por precio en orden descendente, por lo que inicialmente se evalúan las ofertas con mejores precios. Por tanto este algoritmo solo toma decisiones basado en el mejor precio $\boldsymbol{p_i}$ esto implica que:

- El oferente actual (precio más alto) provee la mayor contribución de forma inmediata.
- Las decisiones restantes no dependen de la anterior decisión, se pueden optimizar independientemente.

En esta implementación usamos un algoritmo de ordenamiento para ordenar las ofertas en función del precio en orden descendente, se ve de esta forma:

```
ofertas_ordenadas = sorted(ofertas, key=lambda oferta:
    oferta[1][0], reverse=True)
```

En Python se usa el algoritmo ***Timsort*** para ordenar dicha lista, Timsort combina la ordenación por inserción y la ordenación por mezcla dando como complejidad $\boldsymbol{O(n \log n)}$, dado que este ordenamiento trabaja sobre $\boldsymbol{n}$ elementos nos deja con la misma complejidad.

Por tanto, en el momento contamos con una complejidad dominante de $\boldsymbol{O(n \log n)}$. Despues de ordenar las ofertas, se itera sobre esta nueva lista ordenada para asignar las acciones de forma voraz, de esto se encarga el siguiente bucle:

```
for oferta in ofertas_ordenadas:
    if A == 0:
        break
    if oferta[1][0] >= B:
        acciones_compradas = min(oferta[1][2], A)
        if acciones_compradas >= oferta[1][1]:
            A -= acciones_compradas
            vr += acciones_compradas * oferta[1][0]
```

Cada iteración de este bucle cuenta con un tiempo $O(1)$, ya que estamos haciendo comparaciones, calculos y actualizaciones de valores que corresponden a operaciones básicas. En total iteramos sobre n elementos de la lista ordenada, y por tanto la complejidad total será $O(n)$.

A partir de esto tenemos dos complejidades la de ordenamiento $O(n \log n)$ y la de iteración sobre los elementos ordenados.

La complejidad combinada es la suma de estas partes, pero el ordenamiento domina porque $O(n \log n) > O(n)$ para valores grandes de n. Por lo tanto, la complejidad total del algoritmo es:

$$O(n \log n)$$

En una ejecución con 50 ofertas para una cantidad de acciones $A = 1000000$ con un precio minimo $B = 150$

```
ofertas = [
        (250, 10_000, 20_000),
        (140, 5_000, 25_000),
        (180, 15_000, 30_000),
        (200, 20_000, 40_000),
        (300, 0, 50_000),
        (160, 8_000, 22_000),
        ... # son muchos datos
    ]
```

Obtengo un tiempo de ejecución ínfimo, prácticamente 0.0 (la ejecución es demasiado rápida) y una asignación que resulta optima.

```
Acciones compradas: [(4, 50000), (17, 45000), (27, 40000),
     (37, 50000), (13, 25000), (33, 28000), (46, 30000),
     (19, 27000), (31, 20000), (49, 50000), (0, 20000), (10,
      10000), (24, 35000), (35, 15000), (16, 32000), (30,
     15000), (47, 40000), (26, 26000), (44, 25000), (15,
     18000), (34, 45000), (43, 27000), (7, 35000), (23,
     30000), (39, 40000), (3, 40000), (11, 28000), (22,
     15000), (32, 22000), (38, 30000), (45, 35000), (8,
     20000), (18, 32000)]
```

```
Tiempo de ejecucion: 0.0
Valor total recibido: 242300000
Acciones compradas: [(4, 50000), (17
```

En otro caso de ejecución en donde $A = 1000000$ y $B = 150$ para una cantidad de ofertas $n = 1000$, generando datos aleatorios de la siguiente manera:

```
ofertas = [
    (
        random.randint(20, 400),
        random.randint(0, 20_000),
        random.randint(10_000, 50_000)
    )
    for _ in range(1000)
]
```

Da como resultado un tiempo de ejecución algo más alto, pero igualmente muy rápido:

```
Tiempo de ejecucion: 0.001837015151977539
Valor total recibido: 394246703
```

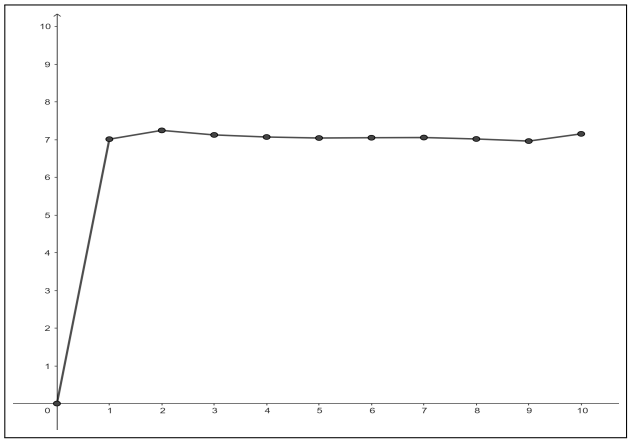
H. Soporte de complejidad computacional

*1) **Solución de fuerza bruta problema de la subasta pública**:* Dentro de lo que es el paradigma de la programación ingenua o de fuerza bruta se tiene que tener en cuenta de que como dijimos en su complejidad lo que se espera puede darnos resultados muy buenos en la parte de buscar una solución optima que nos beneficia, pero por otro lado se complica mas en su tiempo de ejecución, para poder medir sus tiempos hacemos un ejercicio donde A=1000, B=100, n=2, y las ofertas son las siguientes: $< 500, 400, 600 >$, $< 450, 100, 400 >$ y el gobierno $< 100, 0, 1000 >$ con estos datos vamos a ver el tiempo de ejecución que nos da el algoritmo para 10 ejecuciones.

- **Tabla:**

Iteracion	Tiempo
1	7,01041674613952
2	7,24326086044311
3	7,12302041053772
4	7,06792163848876
5	7,04128623008728
6	7,04882073402404
7	7,05390143394470
8	7,01595234870910
9	6,96004986763000
10	7,15144824981689

- **Gráfica:**



lo que podemos observar de esta primera prueba es que los tiempos de ejecución pasan los 7 segundos aun teniendo solo dos oferentes y es que esto se debe a la gran cantidad de combinaciones que tiene que hacer, para ser exactos, en el algoritmo le pedí que me sacara cuantas combinaciones había hecho, siendo un total de 60.561.501 combinaciones y con un tiempo promedio de 7,071607851982110 segundos unos números bastante altos, pero que vamos a comprobar hasta donde podemos estresar mas el algoritmo.

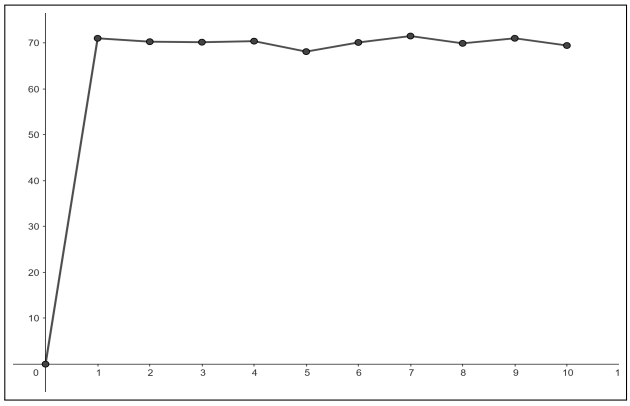
La siguiente prueba se hará con A=800 B=200 n=3, y las ofertas $< 500, 100, 200 >$, $< 450, 100, 150 >$, $< 200, 50, 200 >$ y el gobierno $< 100, 0, 800 >$ con estos datos vamos a ver el tiempo de ejecución que nos da el algoritmo para 10 ejecuciones, lo esperado es que al ser 3

oferentes mas la oferta del gobierno esto nos saque resultados de tiempo de ejecución mucho mas altos.

- **Tabla:**

Iteracion	Tiempo
1	71,0621876716613
2	70,3200123310089
3	70,2122197151184
4	70,4398314952850
5	68,1571490764617
6	70,1537132263183
7	71,5559556484222
8	69,9589543342590
9	71,0647416114807
10	69,4893083572387

- **Gráfica:**



Lo que podemos ver en este es prácticamente lo que se esperaba, la complejidad computacional del algoritmo me indica que entre mas combinaciones existan entre oferentes y gobierno mas grande se va a volver el problema, en este caso el algoritmo hizo 545.238.501 combinaciones y el tiempo promedio de ejecución fue de 70,2414073467254 segundos, casi 10 veces mas que el anterior.

*2) **Solución Dinámica problema de la subasta pública**:*

*3) **Solución Voraz problema de la subasta pública**:* Para realizar las mediciones de tiempos promedios, tomare distintos casos con ofertas generadas al azar, y vamos a ejecutar 50 veces el algoritmo para sacar los tiempos promedios de cada cantidad de ofertas $\boldsymbol{n}$.

En el primer caso tendremos $\boldsymbol{A = 10000}$ $\boldsymbol{B = 450}$ y ofertas que los siguientes valores

$$\boldsymbol{p_i \in \mathbb{Z}, \quad 20 \leq p_i \leq 1000}$$

$$\boldsymbol{m_i \in \mathbb{Z}, \quad 0 \leq p_i \leq 20000}$$

$$\boldsymbol{M_i \in \mathbb{Z}, \quad 10000 \leq p_i \leq 40000}$$

Con valores de $\boldsymbol{n}$ entre 5000 y 95000 los resultados de ejecución fueron los siguientes:

Cantidad de ofertas (n)	Tiempo de ejecución (s)
5000	0.1300547999999253
10000	0.26246420000097714
15000	0.45929590000014286
20000	0.5489888000010978
25000	0.7933214999939082
30000	1.039022400000249
35000	1.2924493000027724
40000	1.5766509999957634
45000	1.8283698999875924
50000	2.099481299999752
55000	2.359104799994384
60000	2.8355879000009736
65000	3.546626199997263
70000	3.3347783999925014
75000	3.7662668999982998
80000	4.417800399998669
85000	5.374115800004802
90000	5.899361400006455
95000	6.724556400004076

Observamos un crecimiento acorde a lo esperado, entre más cantidad de ofertas, más tiempo lleva ordenarlas y por ende más tiempo toma la ejecución del algoritmo, como veremos después no es precisamente una ejecución que tome un tiempo $\boldsymbol{O(n \log n)}$ y esto puede deberse a varios factores.

Ahora con las mismas condiciones anteriores pero para valores de $\boldsymbol{n}$ entre 50000 y 950000

Cantidad de ofertas (n)	Tiempo de ejecución (s)
50000	1.8528631999943173
100000	4.211362800007919
150000	6.814258099999279
200000	9.541652800005977
250000	12.064663199998904
300000	14.4373665000021
350000	17.263603800005512
400000	19.71305819999543
450000	22.70267199999944
500000	25.599663799992413
550000	27.967722899993532
600000	34.84651030000532
650000	43.63037689999328
700000	47.09242169999925
750000	47.504770900006406
800000	45.42212030000519
850000	58.263333599999896
900000	64.41391570000269
950000	53.369315299991285

Continuamos observando la misma tendencia, a medida que la cantidad de ofertas aumenta, aumenta el tiempo de ejecución del algoritmo, el cual es el comportamiento adecuado.

Por último, con las mismas condiciones anteriores pero para valores de $\boldsymbol{n}$ más bajos entre 10 y 960

Cantidad de ofertas (n)	Tiempo de ejecución (s)
10	0.00032730000384617597
60	0.0018326999997952953
110	0.0021938999998383224
160	0.0028334000089671463
210	0.0038409999979194254
260	0.003926899997168221
310	0.005530199996428564
360	0.005935499997576699
410	0.009582600003341213
460	0.006910299998708069
510	0.008769000007305294
560	0.008293499995488673
610	0.012856300003477372
660	0.012865400000009686
710	0.013438200010568835
760	0.013233300007414073
810	0.014419799990719184
860	0.015683700010413304
910	0.015926099993521348
960	0.018464499997207895

Ahora cambiando las condiciones del ejercicio vamos a tener que $\boldsymbol{A = 9500}$ $\boldsymbol{B = 30}$ y ofertas que los siguientes valores

$$\boldsymbol{p_i \in \mathbb{Z}, \quad 20 \leq p_i \leq 1000}$$

$$\boldsymbol{m_i \in \mathbb{Z}, \quad 0 \leq p_i \leq 20000}$$

$$\boldsymbol{M_i \in \mathbb{Z}, \quad 10000 \leq p_i \leq 40000}$$

Para cantidades de ofertas pequeñas de $\boldsymbol{n}$ entre 10 y 960:

Cantidad de ofertas (n)	Tiempo de ejecución (s)
10	0.00019189999147783965
60	0.0006683000101475045
110	0.0012356999941403046
160	0.0017500999965704978
210	0.002396899988525547
260	0.0029084999987389892
310	0.0035271999950055033
360	0.004223600000841543
410	0.006474499998148531
460	0.005784400011179969
510	0.007295999996131286
560	0.007102200004737824
610	0.008092699994449504
660	0.008712100010598078
710	0.01153429999249056
760	0.011930900000152178
810	0.012699300001258962
860	0.013092599998344667
910	0.016119799998705275
960	0.016175099997781217

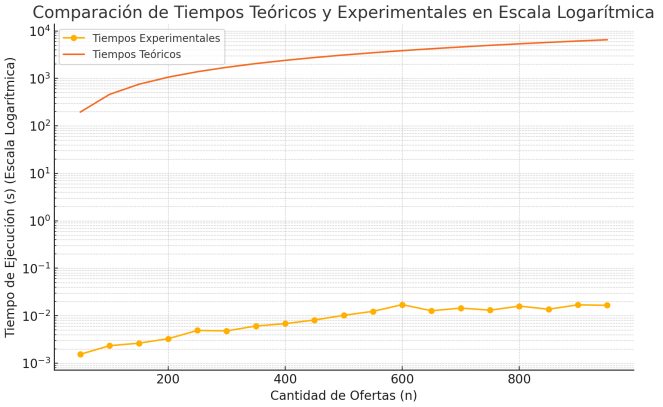
Observamos un crecimiento acorde a la cantidad de ofertas, ademas de que si comparamos con la anterior tabla de valores pequeños, nos damos cuenta de que en este caso al parecer a nuestro algoritmo le toma menos tiempo en promedio hallar la solución optima.

Ahora con las mismas condiciones para valores de $\boldsymbol{n}$ más grandes entre 50000 y 950000

Cantidad de ofertas (n)	Tiempo de ejecución (s)
50000	1.8812771999946563
100000	4.651026199993794
150000	8.194498800003203
200000	11.178894500000752
250000	17.488534500007518
300000	17.71303949999856
350000	15.538597399994615
400000	17.573258599994006
450000	19.759414500003913
500000	22.036872600001516
550000	26.39329190000717
600000	27.16411140000855
650000	28.63515460000781
700000	31.018618500005687
750000	33.22414690000005
800000	35.4484114000079
850000	37.71308689999569
900000	40.90103629999794
950000	44.46773309999844

Nuevamente, si comparamos con la tabla de valores altos para $\boldsymbol{n}$ anterior, nos damos cuenta de que en este caso a nuestro algoritmo le tomo mucho menos tiempo en resolver el problema, esto puede deberse a los valores de las ofertas, ya que como cambiamos dichos valores, se pueden presentar casos en los que la asignación de ofertas es mucho más rápida.

Ahora vamos con las comparaciones de las complejidades teórica y experimental:



En la gráfica logarítmica, las líneas correspondientes a los tiempos teóricos y experimentales siguen una tendencia similar (es decir, ambas parecen crecer aproximadamente al ritmo de $\boldsymbol{O(n \log n)}$), pero están separadas por varias órdenes de magnitud.

La principal razón de esto puede ser que los tiempos teóricos se están basando en la formula matemática que asume un

costo constante por operación, pero esta ignora los detalles de Hardware o la implementación de dicho algoritmo, también debemos considerar que los tiempos experimentales son medidos en segundos en una maquina real y reflejan distintas optimizaciones de compilación, eficiencia de operaciones y otros factores prácticos que pueden afectar dicho tiempo de ejecución.