REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

«Explorando la Distancia entre Cadenas, una Operación a la Vez»

Daniel Quispe

29 de noviembre de 2024

20:51

Resumen

En este informe se observará como la creación de algoritmos de distintos paradigmas es necesario para mejorar la eficiencia de tiempo y espacio. Se verá el caso de Distancia de Edición donde el objetivo principal es ver como observar recurrencia en la exploración de un problema puede ayudar a mejorar dicho problema. Demostrando que buscar como optimizar nuestros algoritmos hace que nuestras busquedas sean más eficientes.

En ese sentido, demostraremos que podemos reducir problemas con complejidad exponencial a problemas con notación cercana a la polinomial. Para ello se crearon dos algoritmos que resulven el mismo caso, uno bajo el paradigma de fuerza bruta y el otro bajo el de programación dinamica. Como resultado, se logro reducir el tiempo de ejecución considerablemente, demostrando que plantear y buscar soluciones para mejorar nuestros algoritmos es ideal.

Índice

1. Introducción

Diseñar programas que resuelvan problemas es una de las razones del gran impacto del campo de la informatica hoy en día, pero dichos programas han tenido que pasar por distintos tipos de algoritmos para poder satisfacer las necesidades actuales. En ese sentido el campo de Analisis y Diseño de algoritmos en Ciencia de la Computación se centra en buscar algoritmos que resuelvan ciertas tareas eficientemente, logrando reducir tiempos de ejecución y espacio auxiliares que puedan llegar a ser optimizados.

De esta manera, uno se puede preguntar ¿Mi algoritmo es eficiente?. LLevandolo a un caso muy conocido como lo es el ordenamiento de datos, un algoritmo que por instinto se le ocurriria a una persona sería; dado una lista de datos, por cada dato en esa lista, revisar los demás para saber si hay un dato más pequeño que el que esta en la primera posición. Dicho algoritmo es conocido por tener una complejidad de $O(n^2)$, pero a través del paradigma de dividir y conquistar se han encontrado algoritmos que han recucido dicho tiempo a $O(nlog(n))$. Este es el gran impacto que tiene la algoritmica en nuestra vida.

En esta ocasión, se hablara sobre 2 paradigmas, el primero se trata de “Fuerza Bruta” el cual busca encontrar todos los posibles caminos hacia una solución. Y el segundo se trata sobre “Programación Dinamica”, la cual busca a través de una serie de pasos, analizar los subproblemas que estos tienen y llegar a una solución una vez encontrado una forma de resolver dichos subproblemas eficientemente.

El algoritmo a analizar se trata del problema “distancia de edición” el cual busca encontrar el minimo coste de operaciones a realizar para que una palabra “s1” se transforme a una palabra “s2”, dichas operaciones son inserción, eliminación, sustitución y además, se agregará a este problema la operación de transponer un elemento. A través de los dos paradigmas dichos anteriormente, se busca comparar el tiempo de ejecución que ambos paradigmas ofrecen al ser implementados en dicho problema. Además, también se verá como la variabilidad de costos puede afectar la optimilidad de dicho problema, dandole un nuevo enfoque al problema ya conocido.

De esta manera, se busca ser consciente que el diseño de un algoritmo si tiene un peso importante en el desarrollo de distintos programas en donde reducir el tiempo de dichas tareas puede tener un impacto significativo en el trabajo que se realiza.

2. Diseño y Análisis de Algoritmos

Dado que las funciones de costo se implementaran en ambos algoritmos, se deja en esta sección.

```
1 Procedure REPLACECOST(a, b)
2     return costo de sustituir a por b
3 Procedure INSERTCOST(b)
4     return costo de insertar b
```

```
1 Procedure DELETECOST(a)
2     return costo de eliminar a
3 Procedure TRANSPOSECOST(a, b)
4     return costo de transponer a por b
```

Todas estas funciones tienen complejidad temporal $O(1)$ ya que es un acceso directo a una matriz o vector de elementos, al igual que su complejidad espacial que es $O(1)$ ya que no usa espacio auxiliar. Tanto las matrices como los vectores se encuentran en variables globales.

2.1. Fuerza Bruta

Para el problema de Distancia de Edición extendida, se a desarrollado un algoritmo bajo el paradigma de Fuerza Bruta, dicho paradigma nos dice que para resolver el problema, se debe revisar todas las posibles soluciones y quedarse con la solución optima que se encuentre, en este caso, dado que que se busca minimización, el valor optimo será el minimo que encuentre el algoritmo.

Bajo dicho concepto, el siguiente algoritmo recibe dos cadenas de strings y el valor de coste obtenido hasta ese momento. En caso de que alguna de las dos cadenas, significa que solo quedará una operación por hacer y esta verá cual es el minimo al terminar dicha ejecución.

Algoritmo 1: Algoritmo bajo paradigma de fuerza bruta para resolver el problema de distancia de edición extendido.

```
1  costeAcumulado ← coste muy grande
2  Procedure FUERZABRUTA(S1, S2, costo)
3      if S1 está vacía then
4          costo ← INSERTCOST(S2[0:])
5          costeAcumulado ← mín(coste, costeAcumulado)
6      else if S2 está vacía then
7          costo ← DELETECOST(S1[0:])
8          costeAcumulado ← mín(coste, costeAcumulado)
9      else
10         FUERZABRUTA(S1[1:], S2[1:], costo + REPLACECOST(S1[0], S2[0]))
11         FUERZABRUTA(S1, S2[1:], costo + INSERTCOST(S2[0]))
12         FUERZABRUTA(S1[1:], S2, costo + DELETECOST(S1[0]))
13         if S1 >1 and S2 >1 and S1[1] == S2[0] and S1[0] == S2[1] then
14             swap(S1[0], S1[1])
15             FUERZABRUTA(S1[1:], S2[1:], costo + TRANSPOSECOST(S1[1], S1[0]))
16             swap(S1[0], S1[1])
```

Dicho algoritmo verá letra por letra hasta que ambos string esten vacios para comparar y encontrar el minimo, si S1 se encuentra vacia antes que S2, esta solo inserta las letras que faltan de S2, por el contrario, si S2 esta vacia antes que S1, las letras sobrantes de S1 se eliminan. En el siguiente ejemplo se muestra

como se va formando el arbol de busqueda, donde el formato de nodos es (S1 - S2 - Minimo encontrado).

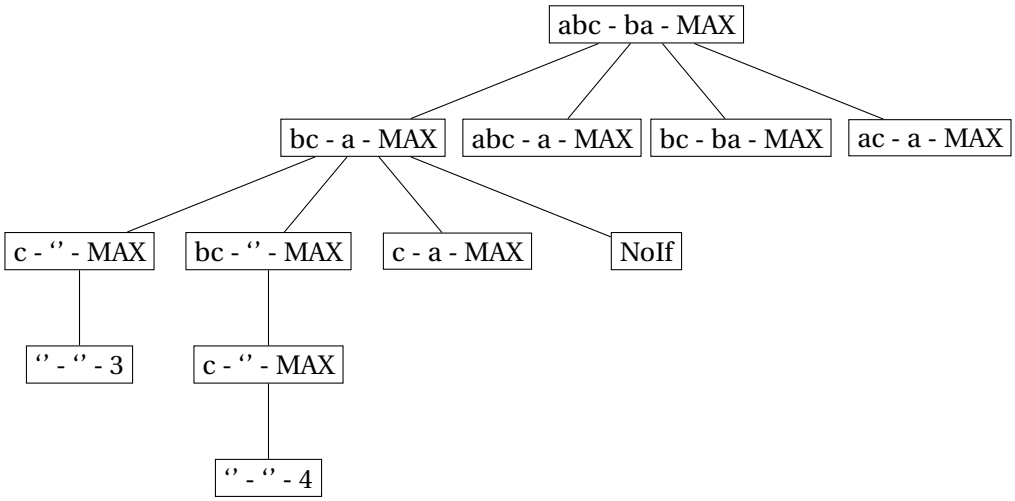

Figura 1: Ejemplo de como el algoritmo va buscando.

En dicho ejemplo, la busqueda al primer nodo es sustitución, al segundo nodo es inserción, el tercer nodo es eliminación, y el cuarto tansposición, solo en caso de que se cumpla las condiciones para entrar al If (por eso la ultima hoja del primer nodo tiene un NoIf), en caso de que un String sea vacio se vera (‘’) y este ejecutara las insercciones o eliminaciones correspondientes para encontrar el valor y ver cual es minimo.

Así, en el caso de que el algoritmo nunca tenga las condiciones para generar un 4to nodo, la complejidad temporal de dicho algoritmo (excluyendo los casos donde S1 y/o S2 es vacio) será de $O(3^{n+m})$. Ahora, si el algoritmo siempre cumple la condiciones, es decir, si se provoca una transposición, la complejidad temporal será de $O(4^{n+m})$ en donde ‘n’ será el largo de S1 y ‘m’ el largo de S2. Pasando a la complejidad espacial, la entrada tiene $O(n+m)$, mientras que el espacio auxiliar es de $O(1)$. Esta ultima es costante ya que no se ocupa ningún tipo de estructura auxiliar para almacenar los datos.

2.2. Programación Dinámica

Para este paradigma, se sigue una secuencia de pasos para poder construir el algoritmo, dichos pasos se estipularán a continuación.

2.2.1. Descripción de la solución recursiva

La solución recursiva se basa en revisar todas las posibles soluciones por cada operación. De esta forma, el arbol de recursión se verá como en el siguiente ejemplo, donde por cada operación realizada, se deben revisar una y otra vez las 4 operaciones hasta llegar a ambos string vacios para obtener el valor de llegar a dicha solución.

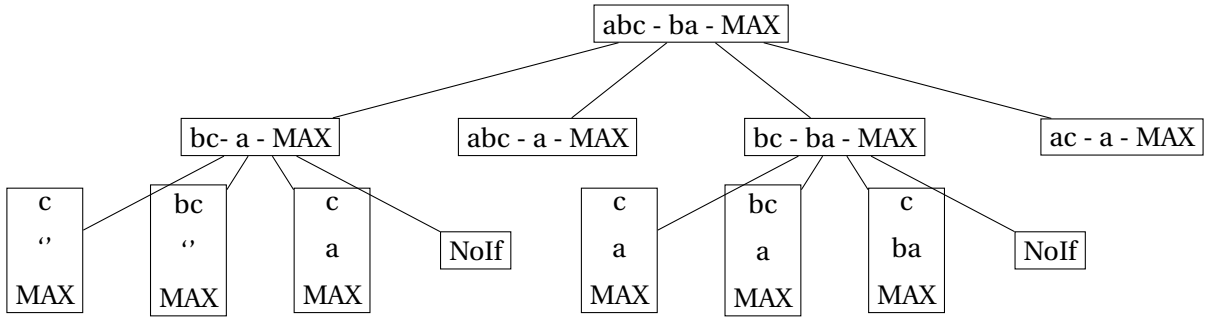

Figura 2: Ejemplo de arbol de recusión.

Pero en dicho ejemplo se puede observar que hay casos donde se repiten casos.

2.2.2. Relación de recurrencia

Como se menciono en el punto anterior, esta busqueda de soluciones tiene puntos en donde se repiten casos, o donde no vale la pena seguir buscando dichas soluciones ya que son más caras que soluciones ya encontradas.

Un caso de recurrencia, se puede observar a continuación.

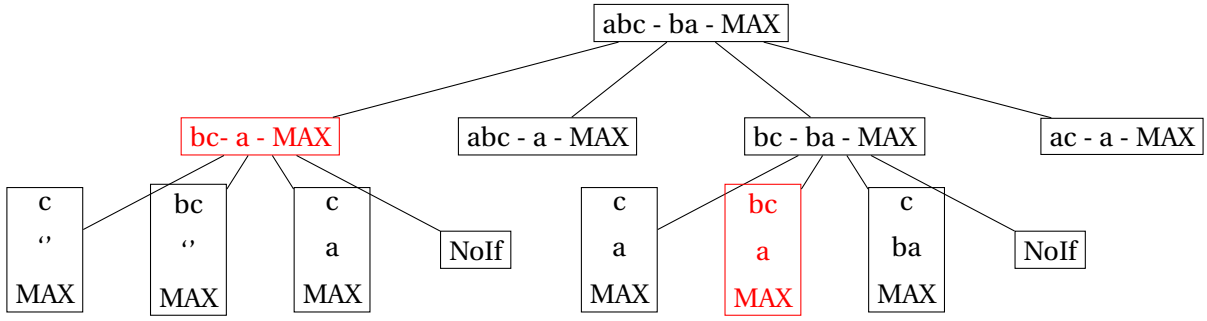

Figura 3: Ejemplo de casos recurrencias.

En donde, se repiten los casos en rojo. Dado que el caso ya se resolvio una vez, este podría ahorrarse en el segundo nodo rojo. Este tipo de casos se reiteran varías veces con distintos nodos, generando valores ya calculadas y que se podrían ahorrar para la creación del algoritmo final.

2.2.3. Identificación de subproblemas

Como se ha estado viendo hasta ahora, hay 4 posibles operaciones que se pueden hacer, cada una de estas operaciones generan subproblemas muy parecidos al problema original. De esta manera el coste minimo es $C_{(i,j)}$ y se han indentificado los siguientes subproblemas:

- Sustituir: $C_{(i,j)} = ReplaceCost(S1[i], S2[j]) + C_{(i-1,j-1)}$
- Insertar: $C_{(i,j)} = InsertCost(S2[j]) + C_{(i,j-1)}$
- Eliminar: $C_{(i,j)} = DeleteCost(S1[i]) + C_{(i-1,j)}$

- Transponer: $C_{(i,j)} = TransposeCost(S1[i-1], S1[i]) + C_{(i-2,j-2)}$

$$C_{(i,j)} = \text{mín} \begin{cases} ReplaceCost(S1[i], S2[j]) + C_{(i-1,j-1)} \\ InsertCost(S2[j]) + C_{(i,j-1)} \\ DeleteCost(S1[i]) + C_{(i-1,j)} \\ TransposeCost(S1[i-1], S1[i]) + C_{(i-2,j-2)} \quad si \quad S1[i-1] = S2[j] \wedge S1[i] = S2[j-1] \end{cases}$$

2.2.4. Estructura de datos y orden de cálculo

Para este algoritmo se utiliza una matriz para almacenar los datos, la matriz se llena de izquierda a derecha y de arriba hacia abajo. Cada casilla se rellena viendo el coste de insertar la j-esima letra en la posición i, el coste de eliminar la i-esima letra, el coste de sustituir (o mantener) la i-esima letra, o el coste de transponer la i-esima letra con su adyacente. A continuación se muestra como itera el algoritmo siguiendo la misma linea de ejemplos mostrados anteriormente (Todos los costes igual a 1).

		a	b	c
	0	1	2	3
b	1			
a	2			

rellenar la primera casilla

en este caso 1 *porque la adjacente es menor*

		a	b	c
	0	1	2	3
b	1	1		
a	2			

La matriz se rellena de tal manera que compara la casilla de arriba, la de la izquierda y la diagonal a esta, y el caso de que se pueda producir una trasposición, se comportará de la siguiente forma:

		a	b	c
	0	1	2	3
b	1	1	2	3
a	2	2		

se puede transponer

en este caso 2 *porque*

		a	b	c
	0	1	2	3
b	1	1	2	3
a	2	2	1	

En dicho caso, el costo se suma al valor diagonal de 2 casillas atras (el valor marcado con rojo) ya que al transponer quedan las dos letras adyacentes en sus posiciones del string objetivo.

2.2.5. Algoritmo utilizando programación dinámica

Siguiendo estos pasos, se puede llegar a un algoritmo bajo el paradigma de programación dinamica. Dada dos cadenas de strings se crea una matriz con tamaño $(n+1)*(m+1)$ siendo 'n' el tamaño del string a modificar y 'm' el tamaño del string objetivo.

A continuación se muestra como se diseñara el algoritmo.

Algoritmo 2: Algoritmo de programación dinamica para resolver el problema de distancia de edición extendida.

```
1  Procedure EDITDISTANCE(S1, S2)
2      if S1 = S2 then
3          return 0
4      else
5          matrix ← matrix[S1_size][S2_size]
6          for i ← 1 to S1_size do
7              for j ← 1 to S2_size do
8                  del ← matrix[i − 1][j] + DELETECOST(S1[i-1])
9                  ins ← matrix[i][j − 1] + INSERTCOST(S2[j-1])
10                 sub ← matrix[i − 1][j − 1] + REPLACECOST(S1[i-1], S2[j-1])
11                 matrix[i][j] ← mín(del, ins, sub)
12                 if S1 >1 and S2 >1 and S1[1] == S2[0] and S1[0] == S2[1] then
13                     matrix[i][j] ← mín(matrix[i][j], matrix[i − 2][j − 2] + TRANSPOSECOST(S1[i-1][i-2]))
14         return matrix[S1_size][S2_size]
```

En este algoritmo el agregar una operación más al problema, no genera ningun impacto importante a la complejidad temporal, ya que este seguira rellenando la matriz de $(n+1)*(m+1)$ elementos. Y para recorrer dicha matriz se requieren dos ciclos for anidados que ocupan una complejidad temporal de $O(n*m)$, siendo 'n' el tamaño del string a modificar y 'm' el tamaño del string objetivo. La complejidad espacial de la entrada será de $O(n+m)$, mientras que el espacio auxiliar como se menciono, será la matriz y tendrá complejidad espacial de $O(n*m)$

3. Implementaciones

3.1. Fuerza Bruta

Para el algoritmo de fuerza bruta, se inicializan las tablas de costo y se piden por consola que se ingresen ambas palabras, luego, ambas palabras se mandan a la función “fuerzaBruta” donde se buscara recursivamente hasta lo más profundo del arbol y revisa cual es el minimo. Dicho algoritmo se encuentra en este enlace.

3.2. Programación Dinámica

En este algoritmo se llama a la función “edit_distance”, se rellena una matriz con datos del minimo camino que se puede hacer para llegar a dicha casilla. Dicha matriz de $(n+1)*(m+1)$ se rellena de izquierda a derecha y de arriba hacia abajo y retorna el ultimo valor obtenido en la matriz (valor de la esquina inferior derecha) que es el valor optimo. El algoritmo se encuentra en este enlace.

3.3. Funciones auxiliares

En ambos programas se repetian funciones, así que se crearon dos archivos para obtener los datos sin necesidad de implementar los mismos codigos. Cabe resaltar que ambos algoritmos usan la misma cantidad de archivos para su compilación.

El primer archivo llamado “costos.hpp” son las funciones con las cuales se pueden obtener los costos de las distintas operaciones, todas estas funciones tienen complejidad espacial y temporal de $O(1)$ ya que no se guarda nada y todo es constante. Mientras que la función para inicializar la matriz a rellenar tiene complejidad $O(n+m)$ ya que tiene que recorrer a través de un par de ciclos for para rellenar la primera fila y columna de la matriz. El programa con dichas funciones se encuentra en este enlace.

El segundo archivo llamado “funtion.hpp” contiene funciones de inicialización, como lo son los de rellenar las matrices y vectores con los datos entregados a través de la carpeta “data”. Todas estas funciones tienen complejidad $O(1)$ ya que no varían según alguna entrada, estas son fijas en 26 datos. El programa con dichas funciones se encuentra en este enlace.

4. Experimentos

Los experimentos se realizaron a través de una arquitectura la cual posee Intel Core i5-10400, 2.90GHz, 16GB RAM DRR4, almacenamiento de SSD NVMe. Todo se realizo a través del sistema operativo Windows 10 y se compilo a través del subsistema operativo de windows para linux (WSL) Ubuntu 22.04.5 LTS, compilador g++ 11.4.0.

Los resultados obtenidos se encontrarán a continuación.

4.1. Dataset (casos de prueba)

Los casos de prueba se encuentran en este enlace. En este archivo se encuentran los 10 casos de pruebas usados, y se comentarán los 5 casos más importantes, bajo distintos costos asociados.

El primer caso evaluado es:

- S1 = intention y S2 = execution
- costo_sub(a, b) = 1 si $a \neq b$, y 0 $a = b$.
- costo_trans(a, b) = 1 para transponer los caracteres adyacentes a y b.
- costo_ins(b) = 1 para cualquier carácter b.
- costo_del(a) = 1 para cualquier carácter a.
- Salida esperada: 5

Un camino podría ser:

$i \rightarrow e$ sustituir +1	$n \rightarrow c$ sustituir +1
$n \rightarrow x$ sustituir +1	$+u$ insertar +1
$-t$ eliminar +1	total = 5

Cambiando los costos de operaciones quedaría como se nuestra a continuación:

- costo_sub(a, b) = 2 si $a \neq b$, y 0 $a = b$.
- costo_trans(a, b) = 1 para transponer los caracteres adyacentes a y b.
- costo_ins(b) = 5 para cualquier carácter b.
- costo_del(a) = 3 para cualquier carácter a.
- Salida esperada: 10

El camino sería:

$i \rightarrow e$ sustituir +2	$e \rightarrow c$ sustituir +2
$n \rightarrow x$ sustituir +2	$n \rightarrow u$ sustituir +2
$t \rightarrow e$ sustituir +2	total = 10

Los casos donde alguno de los strings es vacio se verán así (Se ocuparan los ultimos costes de operaciones utilizado):

- S1 = ‘’ y S2 = holamundo
- salida esperada: 45

Esto porque solo se hacen operaciones de inserción. Como son 9 caracteres, quedaría:
$9 * 5 = 45$ (5 porque es el valor de inserción).

- S1 = chaomundo y S2 = ‘’
- salida esperada: 27

Esto porque solo se hacen operaciones de eliminación. Como son 9 caracteres, quedaría:
$9 * 3 = 27$ (3 porque es el valor de eliminar).

Otro caso de interes, es cuando los strings no tienen el mismo largo, en dicho caso, la sitaucíon no cambia mucho (Se ocuparan los ultimos costes de operaciones utilizado):

- S1 = kitten y S2 = sitting
- salida esperada: 9

El camino sería:
$k \rightarrow s$ sustituir +2
$e \rightarrow i$ sustituir +2
$+g$ insertar +5
total = 9

Y el ultimo caso sería que es util una transposición (Se ocuparan los ultimos costes de operaciones utilizado):

- S1 = abababcabab y S2 = bababacbaba
- salida esperada: 5

Caso sin transposición:
$-a$ eliminar +3 (posición 1)
$+a$ insertar +5 (posición 6)
$-a$ eliminar +3 (posición 8)
$+a$ insertar +5 (posición 11)
total = 16

Caso con transposición:
$a \rightarrow b$ transponer +1 (posición 1 con 2)
$a \rightarrow b$ transponer +1 (posición 3 con 4)
$a \rightarrow b$ transponer +1 (posición 5 con 6)
$a \rightarrow b$ transponer +1 (posición 8 con 9)
$a \rightarrow b$ transponer +1 (posición 10 con 11)
total = 5

En este ultimo caso, dado que el coste de transposición era mucho menor que las otras operaciones, era mucho más eficiente hacer la transposición. Es por ello que tener esta operación es util en ciertos casos. Tener esta cuarta operación amplia la posibilidad de encontrar distintos optimos.

Los otros 5 casos de prueba serán mencionados en los resultados que estos obtuvieron.

4.2. Resultados

Para la toma de datos, se ha decidido tomar datos que cumplan con ciertas caracteristicas. Los primeros 5 casos, son casos comunes que el algoritmo debería poder resolver sin problemas, mientras que los siguientes 5 se tratan de probar el tiempo que estos algoritmos tardan en resolver dichos casos de prueba para comparar el tiempo de ambos algoritmos.

Para la obtención de dichos datos, es necesario colocar distintos archivos de texto (con extensión .txt) en la carpeta 'datasets' bajo el nombre de inputN°.txt, donde N° es un numero cualquiera del 1 al 10 (el programa te pide ingresar un numero, así que mientras el archivo con dicho numero este dentro de la carpeta igual se puede acceder a él). Estos archivos tiene solo 2 lineas (es necesario que el archivo si o si

tenga dos lineas, independiente si esta vacia o no). Una vez escogido el archivo, se ejecutara el algoritmo y entregará por pantalla el tiempo que tardo el algoritmo y el coste de edición.

La siguiente tabla tiene el tiempo que cada algoritmo tardo con los siguientes pares de palabras (Como ya se menciono, si una palabra esta vacía se identificará con ‘’).

String 1 **String 2**	**Tamaño entrada** **(n + m)**	**Tiempo de** **Fuerza Bruta**	**Tiempo de** **Programación Dinamica**
intention execution	18	0.06997	0.00001
‘’ holamundo	9	0.00000	0.00000
chaomundo ‘’	9	0.00000	0.00002
kitten sitting	13	0.00104	0.00001
abababcabab bababacbaba	22	2.76720	0.00001
dasdoibd oiassadubihn	20	0.23500	0.00001
estoycansado estoycantando	25	28.23300	0.00003
asdfghjklqwertyu mnbvcxzpoi	26	27.28609	0.00001
primerapalabramuy segundapalabramas	36	-	0.00002
primerapalabramuylargas segundapalabramasgrande	46	-	0.00006

A través de la tabla anterior se puede observar que la implementación del algoritmo bajo el paradigma de programación dinamica es mucho más rápido, ya que como se explico anteriormente, la complejidad del algoritmo de fuerza bruta es de $O(3^{n+m})$ mientras que la de programación dinamica es de $O(n * m)$, dicho cambio en la complejidad se puede ver reflejada en los tiempos obtenidos.

Como se puede aprecíar en la fig. 4 a medida que el tamaño de la entrada va aumentando, el tiempo de ejecución del algoritmo de fuerza bruta se dispara exponencialmente, siendo imposible para la arquitectura utilizada para los experimentos lograr medir el tiempo del tamaño más grande de entrada.

Pero el tiempo no es el unico factor a evaluar, el espacio que estos algoritmos ocupan también es importante. Como se puede ver en la siguiente tabla, el algoritmo de fuerza bruta se mantiene constante a medida que crece el tamaño de entrada, a diferencia del algoritmo de programacióm dinamica que va aumentando su espacio a medida que crece el tamaño de entrada. Esto ultimo se debe al espacio auxiliar que ocupa dicho algoritmo el cual es $O(n * m)$.

En la fig. 5 se puede apreciar como el espacio aumenta a medida que el tamaño de entrada crece.

String 1 **String 2**	**Tamaño entrada** **(n + m)**	**Espacio de** **Fuerza Bruta (bytes)**	**Espacio de Programación** **Dinamica (bytes)**
intention execution	18	131519	132199
‘’ holamundo	9	131492	131596
chaomundo ‘’	9	131519	131803
kitten sitting	13	131510	131934
abababcabab bababacbaba	22	131525	132437
dasdoibd oiassadubihn	20	131516	132252
estoycansado estoycantando	25	131528	132624
asdfghjklqwertyu mnbvcxzpoi	26	131789	132836
primerapalabramuy segundapalabramas	36	-	133561
primerapalabramuylargas segundapalabramasgrande	46	-	134797

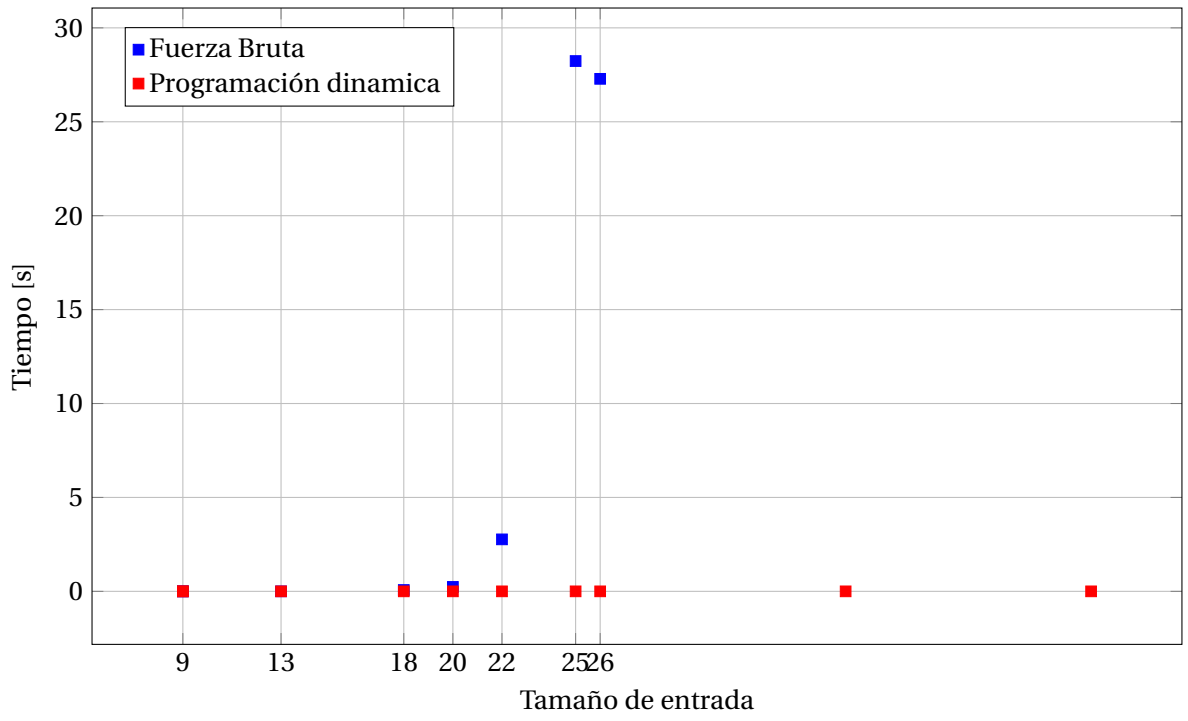

Figura 4: Tiempo de ejecución vs Tamaño de entrada.

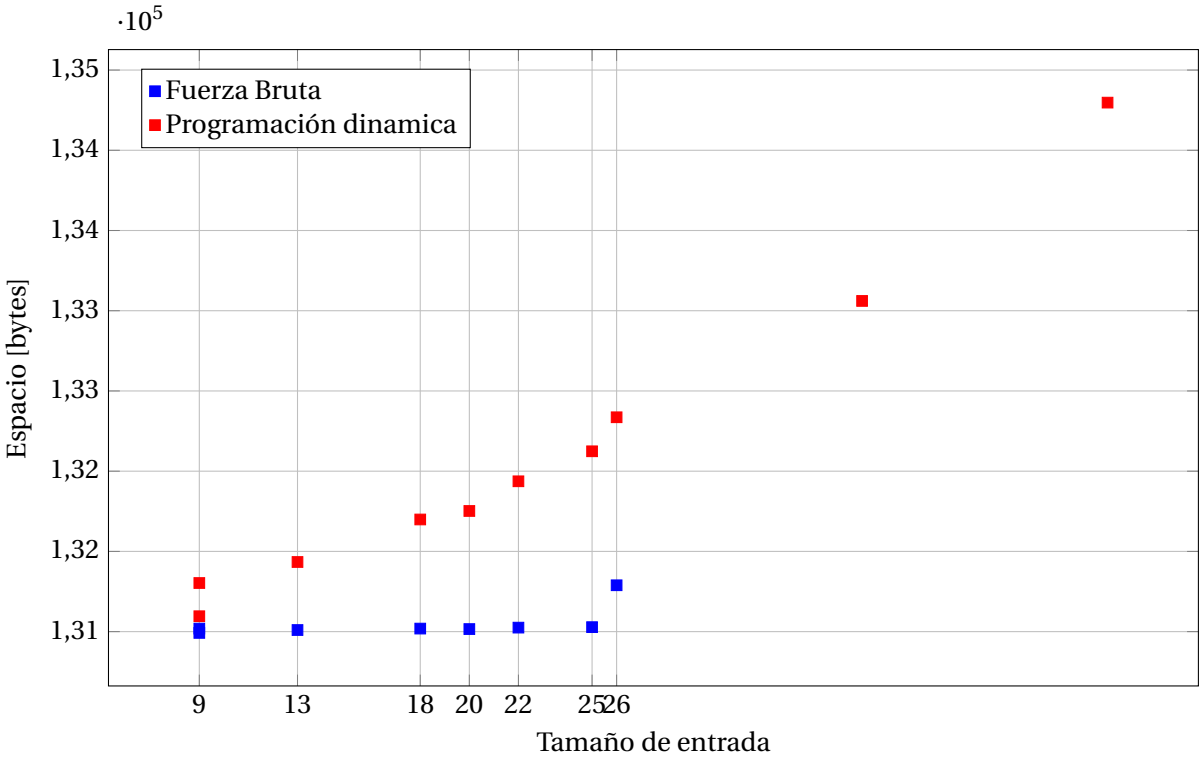

Figura 5: Espacio utilizado vs Tamaño de entrada.

5. Conclusiones

A través de todo lo explicado, se ha podido observar como la manera en que se programa algunos problemas, estos pueden ser mucho más eficientes a la hora de ahorrar tiempo. Pero como se pudo observar, dicho manejora de tiempo igual impacto en otro ambito, en este caso el uso de mayor espacio. Pero dado que la disminución de tiempo es mucho más significativa que el uso de espacio, este ultimo no es un impacto tan importante para para la comparativa entre algoritmos.

De esta manera, se puede demostrar que a través de una buena planificación y lograr encontrar ciertas caracteristicas, se puede lograr una mejora de eficiencia, logrando reducir la complejidad de $O(3^{n+m})$ o $O(4^{n+m})$ a $O(n*m)$ como se pudo demostrar en la fig. 4 solo teniendo que sacrificar espacio, el cual como se pudo ver a través de la fig. 5, esta no es un impacto tan significativo como lo es la complejidad temporal del primer algoritmo.

Concluyendo, la busqueda de la eficiencia y optimización, si son necesarios hoy en día para mejorar el tiempo de respuesta de lo que estemos tratando de realizar, así como se demostro en este experimento.