# INFORME TAREA 2 y 3
# ALGORITMOS Y COMPLEJIDAD

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

Diego Sierra

19 de noviembre de 2024

**Resumen**

*Este informe explora algoritmos para calcular la distancia de edidicion extendida (agregando transposicion) implementando 2 algoritmos, fuerza bruta y programación dinámica, comparando sus eficiencias reales y enfatizando la importancia de seleccionar el algoritmo correcto segun las necesidades del problema a resolver, se exploran las complejidades computacionales, estrategias de resolución y rendimiento, se estudian estos algoritmos con diferentes datasets, las operaciones ocupadas son inserciones, eliminaciones, sustituciones y transposiciones.*

## Índice

# 1. Introducción

El problema de edicion de caracteres es un clasico problema en ciencias de la computacion, en el cual se busca encontrar la cantidad minima de operaciones para transformar una cadena de caracteres en otra. Este problema es de gran importancia en la vida cotidiana, ya que se puede aplicar en la corrección de errores de texto, en la comparación de archivos, analisis de similitud de texto, etc.

Este informe esta enfocado en la implementación de 2 algoritmos para la resolución de estos problemas. En particular, se busca encontrar la solucion de, dada 2 cadenas de caracteres cual es el costo minimo de edicion de 1 de estas cadenas para que las cadenas sean iguales. La investigacion de centra en analizar los distintos algoritmos para poder determinar cual es el mas eficiente en terminos de tiempo y espacio. Acaso programación dinámica es mejor que fuerza bruta en todos los casos ? o hay casos en los que fuerza bruta es mejor que programación dinámica ? Estas son algunas de las preguntas que se buscan responder con esta investigacion, la respuesta de estas preguntas es de gran importancia ya que nos permitira saber como se comportan estos algoritmos en diferentes situaciones y asi poder elegir el algoritmo mas adecuado para cada caso.

Los algoritmos a implementar son, de programación dinámica y de fuerza bruta.

Las operaciones permitidas para este estudio seran: inserción, eliminación, sustitucion y transposicion de caracteres. Para la implementación de los algoritmos se utilizó el lenguaje de programación de C++.

Es importante que tengan a mano el codigo fuente de este estudio para poder seguir la explicación de los algoritmos, poder entender como se implementan, como funcionan y por si se les presenta dudas sobre codigo en particular. https://github.com/Daspssj/Tarea2_3

# 2. Diseño y Análisis de Algoritmos

## 2.1. Fuerza Bruta

La solucion diseñada para el algoritmo de fuerza bruta se enfoca en recorrer todas las posibles combinaciones de las operaciones de inserción, eliminacion, sustitución y transposición, comparando la cadena de strings `S1` y `S2` para encontrar la distancia minima de edicción extendida, modificando unicamente una de estas 2 cadenas en cada iteración, en nuestro caso solo se modifica la primera cadena `S1`.

Como hay que expresar las complajidades en terminos de las longitudes de las cadenas de entrada $S1$ y $S2$, se tiene que n y m son las longitudes de las cadenas $S1$ y $S2$ respectivamente y se tiene que la complejidad temporal de este algoritmo es de $O(4^{\text{máx}(n,m)})$ y la complejidad espacial es de $O(\text{máx}(n,m))$.

La transposiciones y costos variables impactan significativamente en la complejidad en el caso de las transposiciones se tiene que la complejidad temporal aumenta a $O(4^{\text{máx}(n,m)})$ siendo la complejidad temporal sin la transposición de $O(3^{\text{máx}(n,m)})$ esto se debe a que en el peor caso se tiene una mayor cantidad de llamadas recursivas, en cambio la complejidad espacial con la transposición es la misma que la vista anteriormente.

Para el ejemplo de ejecucionse tienen las cadenas $S1$ = "kittenz $S2$ = "sitting", se tiene que la distancia de edición mínima es de 1, y se puede obtener de la siguiente manera:

- **Sustituir 'k' por 's':** k $\rightarrow$s
- **Insertar 'g' al final de la cadena:** $\rightarrow$g
- **Sustituir 'e' por 'i':** e $\rightarrow$i

**Algoritmo 1:** Fuerza Bruta para calcular la distancia mínima de edición extendida.

```
1  Procedure FuerzaBruta(S1, S2)
2      n ← longitud de S1
3      m ← longitud de S2
4      if n = 0 then
5          return Costo de insertar todos los caracteres de S2
6      else if m = 0 then
7          return Costo de eliminar todos los caracteres de S1
8      costo_sust ← CostoSust(S1[n-1], S2[m-1]) + FuerzaBruta(S1[0:n-1], S2[0:m-1])
9      costo_elim ← CostoElim(S1[n-1]) + FuerzaBruta(S1[0:n-1], S2)
10     costo_inser ← CostoInser(S2[m-1]) + FuerzaBruta(S1, S2[0:m-1])
11     costo_trans ← ∞
12     if n > 1 and m > 1 and S1[n − 1] = S2[m − 2] and S1[n − 2] = S2[m − 1] then
13         costo_trans ← CostoTrans(S1[n-2], S1[n-1]) + FuerzaBruta(S1[0:n-2], S2[0:m-2])
14     return Minimo costo entre todas las opereaciones
```

## 2.2. Programación Dinámica

### 2.2.1. Descripción de la solución recursiva

Este algoritmo busca calcular el costo mínimo de transformar una cadena de caracteres $S1$ en otra cadena de caracteres $S2$. Para ello, se consideran las siguientes operaciones:

- **Inserción:** Insertar un carácter a la cadena $S1$.
- **Eliminación:** Eliminar un carácter de la cadena $S1$.
- **Sustitución:** Reemplazar un carácter de la cadena $S1$ por otro que le correspon a $S2$.
- **Transposición:** Intercambiar dos caracteres consecutivos de la cadena $S1$ y $S2$.

El algoritmo se enfoca en comparar el costo de realizar cada una de estas operaciones para cada caracter para asi elejir la operación que minimice el costo total de transformar $S1$ en $S2$.

### 2.2.2. Relación de recurrencia

Sea dp[i][j] el costo mínimo de transformar los primeros i caracteres de S1 en los primeros j caracteres de S2, tenemos que la siguiente relación de recurrencia:

$$dp[i][j] = \begin{cases} j \cdot \text{costo\_inser}(s2[j-1]), & \text{si } i = 0 \\ i \cdot \text{costo\_elim}(s1[i-1]), & \text{si } j = 0 \\ \text{mín}\Big(dp[i][j-1] + \text{costo\_inser}(s2[j-1]), & \text{(insertar)} \\ dp[i-1][j] + \text{costo\_elim}(s1[i-1]), & \text{(eliminar)} \\ dp[i-1][j-1] + \text{costo\_sust}(s1[i-1], s2[j-1]), & \text{(sustituir)} \\ dp[i-2][j-2] + \text{costo\_transpos}(s1[i-1], s2[j-1]) & \text{(transponer)}\Big) \end{cases}$$

Como se puede obeservar en la relación de recurrencia, se consideran los casos base cuando $i = 0$ y $j = 0$, que corresponden a los costos de insertar y eliminar todos los caracteres de $S2$ y $S1$, respectivamente, y tambien se incluye la operación de transposición si se cumplen las condiciones necesarias.

### 2.2.3. Identificación de subproblemas

La soluicon que propone este algoritmo de transformar una cadena de caracteres $S1$ en otra cadena de caracteres $S2$ (entregando los costos asociados) se puede dividir en subproblemas más pequeños, que corresponden a transformar subcadenas de $S1$ en subcadenas de $S2$, se busca transformar los prefijos inmediatos de $S1$ y $S2$ en cada iteración para luego sumar el costo de la operación que minimice el costo total de transformar $S1$ en $S2$.

#### 2.2.4. Estructura de datos y orden de cálculo

Para resolver este problema utilizando programación dinámica, se propone utilizar una matriz $dp$ de tamaño $(n+1) \times (m+1)$,donde $n$ y $m$ son las longitudes de las cadenas $S1$ y $S2$, respectivamente. El programa utiliza programación dinamica con un enfoque de Bottom-Up. En la matriz se inicializan los valores de los casos base de manera que la primera fila es el costo de insertar caracteres en $S1$ y la primera columna respresenta el costo de eliminar caracteres de $S1$,luego se recorren las filas y columnas de la matriz para calcular el costo mínimo de transformar los prefijos de $S1$ y $S2$ en cada iteración, se llenan los valores de la matriz de izquierda a derecha y de abajo hacia arriba, de manera que al final de la ejecución el valor de $dp[n][m]$ corresponderá al costo mínimo total.

#### 2.2.5. Complejidades

La complejidad temporal y espacial de este algoritmo son las mismas y es de $O(n \times m)$, donde $n$ y $m$ son las longitudes de las cadenas $S1$ y $S2$, respectivamente.

#### 2.2.6. Ejemplo de ejecución

Para las cadenas $S1$ = "kittenz $S2$ = "sitting", se tiene que la distancia de edición mínima es de 1, y se puede obtener de la siguiente manera:

- **Sustituir 'k' por 's':** k $\rightarrow$s
- **Insertar 'g' al final de la cadena:** $\rightarrow$g
- **Sustituir 'e' por 'i':** e $\rightarrow$i

| | | s | i | t | t | i | n |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| k | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| i | 2 | 2 | 1 | 2 | 3 | 4 | 5 |
| t | 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| t | 4 | 4 | 3 | 2 | 1 | 2 | 3 |
| e | 5 | 5 | 4 | 3 | 2 | 3 | 4 |
| n | 6 | 6 | 5 | 4 | 3 | 4 | 3 |

Cuadro 1: Matriz de costos para transformar la cadena "kitten.[en] "sitting".

#### 2.2.7. Impacto de las transposiciones y costos variables

Las transpocisiones y los costos variables para este caso no impactan en absoluto a la complejidad temporal y espacial, por lo que la complejidad temporal y espacial se mantiene en $O(n \times m)$.

### 2.2.8. Algoritmo utilizando programación dinámica

**Algoritmo 2:** Programación Dinámica para calcular la distancia mínima de edición.

```
1  Procedure PROGDINAMICA(S1, S2)
2      n ← longitud de S1
3      m ← longitud de S2
4      Crear matriz dp de tamaño (n+1) × (m+1)
5      for i ← 0 to n do
6          dp[i][0] ← costo de eliminar todos los caracteres hasta i en S1
7      for j ← 0 to m do
8          dp[0][j] ← costo de insertar todos los caracteres hasta j en S2
9      for i ← 1 to n do
10         for j ← 1 to m do
11             costo_inser ← dp[i][j−1] + COSTOINSER(S2[j-1])
12             costo_elim ← dp[i−1][j] + COSTOELIM(S1[i-1])
13             costo_sust ← dp[i−1][j−1] + COSTOSUST(S1[i-1], S2[j-1])
14             costo_trans ← ∞
15             if i > 1 and j > 1 and S1[i−1] = S2[j−2] and S1[i−2] = S2[j−1] then
16                 costo_trans ← dp[i−2][j−2] + COSTOTRANS(S1[i-2], S1[i-1])
17             dp[i][j] ← mín(costo_inser, costo_elim, costo_sust, costo_trans)
18     return dp[n][m]
```

# 3. Implementaciones

- **Estructura de archivos:** El archivo donde se encuentra las implementacines de los algoritmos es Algoritmos.cpp , en este archivo se encuentran las implementaciones de los algoritmos de fuerza bruta y programación dinámica, como la funcion del main que se encarga de leer todos los datos de entrada (esto se refiere a las tablas de costos de 1 y 2 dimensiones junto con los inputs de los datasets que contienen los 2 strings a comparar y editar) y llamar a los algoritmos correspondientes.

  En esta carpeta (Tarea 2 y 3) tambien se encuentran los costos de las operaciones de inserción, eliminación, sustitución y transposición, como tambien las distintas datasets que se utilizaron para probar los algoritmos.

  Tambien se encuentra el archivo de Creardataset.cpp que se encarga de crear los datasets que se utilizaron para probar los algoritmos.

- **Funciones:** Las funciones mas importantes son dynamic_progra() y brute_force(), que corresponden a las implementaciones de los algoritmos de programación dinámica y fuerza bruta, respectivamente.

# 4. Experimentos

## 4.1. Infraestructura utilizada

Para la realización de los experimentos, se utilizó un computador con las siguientes características:

- **Procesador:** Intel Core i7-14700KF, 5.6 GHz
- **Memoria RAM:** 32 GB DDR5
- **Almacenamiento:** 2 TB SSD NVMe M.2
- **Sistema Operativo:** Windows 10 Pro
- **Compilador:** g++ 11.4.0
- **Modo de compilacion** g++ -Wall {Archivo}.cpp -o test
- **Librerías:** C++ Standard Library
- **Entorno de Desarrollo:** Ubuntu 11.4.0

## 4.2. Datasets

Los datasets utilizados para las pruebas fueron 6, y son de 32 palabras de largo variablem mas no aleatorio, estos datasets fueron generados de menera automatica y se guardaron en archivos de texto, cada dataset tiene un nombre, y fueron las siguientes:

- **dataset_mismotamano.txt:** Contiene palabras de misma longitud que varia entra 1 y 15 caracteres.
- **dataset_s1_vacia.txt:** Contiene palabras donde S1 es vacia y S2 tiene longitud variable.
- **dataset_s2_vacia.txt:** Contiene palabras donde S2 es vacia y S1 tiene longitud variable.
- **dataset_s1>s2.txt:** Contiene palabras donde S1 es mayor o igual que S2.
- **dataset_s1<s2.txt:** Contiene palabras donde S2 es mayor o igual que S1.
- **dataset_transposicion.txt:** Contiene palabras transpuestas.

La importancia de estos datasets radica en que se pueden probar los algoritmos con distintos casos de prueba, y se pueden verificar si los algoritmos son capaces de resolverlos de manera correcta en varios casos, esto nos puede dar una idea de la eficiencia y efectividad de los algoritmos, lo que se busca probar con estos datasets es que los algoritmos sean capaces de resolver problemas donde las cadenas sean de distinto e igual tamaño, que pueda resolver problemas donde los dos strings o solamente uno de ellos este vacio y por ultimo que pueda resolver problemas donde las cadenas de caracteres esten transpuestas.

### 4.3. Resultados

Para el analisis de los resultados es importante notar que en el eje y estan los tiempos de ejecucion en microsegundos y en el eje x se tiene el largo de las cadenas de entrada, estos graficos estan dentro del archivo de la tarea, el archivo el archivo en cuestion se llama graficos.ipynb, las graficas no son generadas de manera automatica por lo que estos son los resultados de un promedio de 3 pruebas para cada datasets, favor de observar el codigo y en caso de replicar el experimento identificar los datos a cambiar para asi poder ver sus respectivas graficas, el programa esta hecho en jupyther y es muy intuitivo de usar para cambiar los resultados experimentales puede ver el codigo aca https://github.com/Daspssj/Tarea2_3/blob/main/graficos.ipynb, para generar los datos en caso de querer replicar el trabajo favor de compilar y ejecutar el archivo de Crear_datasets.cpp https://github.com/Daspssj/Tarea2_3/blob/main/Crear_datasets.cpp y para generar sus propios resultados se tiene que compliar y ejecutar el archivo de Algoritmos.cpp https://github.com/Daspssj/Tarea2_3/blob/main/Algoritmos.cpp el programa implementado es bastante amigable con el ususario por lo que no sera dificil usarla, a continuacion se presentan los resultados obtenidos de las pruebas realizadas:

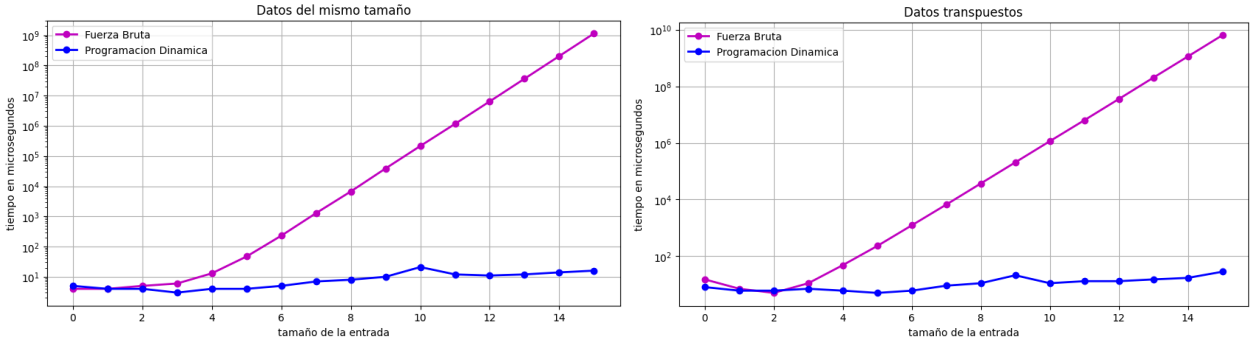


Figura 1: Tiempos de ejecucion vs largo de las cadenas

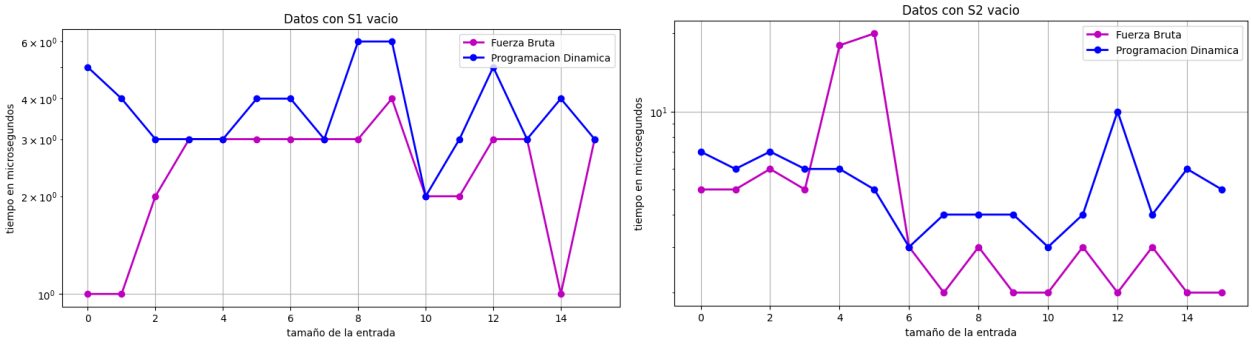


Figura 2: Tiempos de ejecucion vs largo de las cadenas

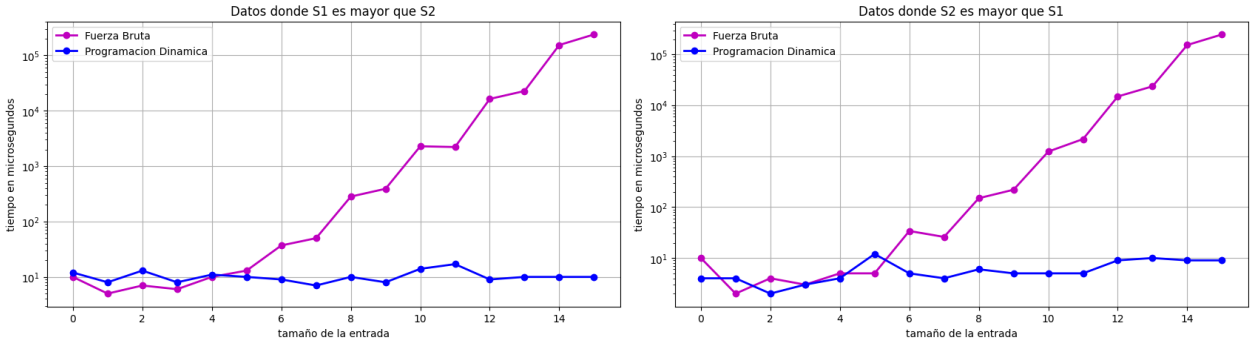


Figura 3: Tiempos de ejecucion vs largo de las cadenas

Como podemos observar en los primeros 2 graficos correspondientes a la figura 1 se puede observar que los tiempos de ejecucion de cada algoritmo son muy distintos entre si y esto se debe a las complejidades temporales de los algoritmos, se puede ver que el agoritmo de fuerza bruta crece de manera exponencial mientras que programacion dinamica se mantiene en un rango de valores aceptable, luego para los algunos de los strings vacios se ve que Fuerza bruta tiene un tiempo de ejecucion mas bajo que el de programacion dinamica aunque no mucho igualmente se puede ver en los graficos de la figura 2 y por ultimo para los strings de distinto tamaño se puede ver que el algoritmo de fuerza bruta tiene un tiempo de ejecucion mas alto que el de programacion dinamica, sin embargo estos tiempos no son tan altos como en el caso de los strings de mismo tamaño ya que solo se transpone una cadena, esto quiere decir que no se tiene que recorrer todas las posibles combinaciones entre insertar, eliminar, sustituir o transponer.

Por otro lado tenemos el uso de memoria de los algortimos, donde el algoritmo de fuerza bruta tiene un uso de memoria de $O(n)$, lo que significa que su uso de memoria es lineal en donde $n$ es el tamaño de la cadena de entrada, por otro lado tenemos el algoritmo de programacion dinamica el cual su uso de memoria es el tamaño en bytes de la matriz que se crea para almacenar los resultados de las subcadenas, por lo que su uso de memoria es de $O(n \times m)$ donde $n$ y $m$ son los tamaños de las cadanas de entrada.

Nota: Es importante tener en cuenta el uso del cache para el primer resultado de fuerza bruta y programacion dinamica, ya que se tiene que en algunos casos el primer tiempo de ejecucion de los algoritmos son raramente mas altos que los demas primeros datos.

## 5. Conclusiones

Tras el diseño, analisis y experimentación de los algoritmos de fuerza bruta y programación dinámica, se pueden concluir varios puntos importantes de resaltar, como que la complejidad temporal de la fuerza bruta es exponecial, lo que se traduce en un tiempo de ejecucion muy alto para grandes volumenes de datos, lo que se puede observar en los graficos proporcionados, por otro lado tenemos que la programación dinámica se comporta de manera mucho mas eficiente para volumenes muy grandes de datos, teniendo estas 2 caracteristicas en cuenta, se puede concluir que la programación dinámica es la mejor opcion para resolver este tipo de problemas con grandes volumenes de datos, o sea, para palabras muy largas (llevado a la vida real), sin embargo se pudo ver que para las palabras vacias y/o una vacia y la otra no, la fuerza bruta es mas eficiente que la programación dinámica, por lo que se tiene que programación dinámica es la mejor opcion para palabras de gran tamano ( >10 caractares) y la fuerza bruta es la mejor opcion para comparaciones donde una de las palabras es vacia o ambas palabras son vacias.Por otro lado se tiene que, si se busca utulizar un algortimo que no consuma tanta memoria (o nula) es altamente recomendable la fuerza bruta ya que como y hemos visto, la programación dinámica tiene una complejidad espacial mas alta que la ya dicha.Tambien se puede observar que se conserva la verasidad de los algoritmos ya que ambos algoritmos fueron capaces de encontrar una solucion a los diversos escenarios planteados.

La investigación de este problema nos permitió conocer y entender la importancia de los distintos algoritmos en la resolución del problema de la distancia de edición, tambien quedo claro la importancia de seleccionar el algoritmo correcto segun las necesidades del problema a resolver.

## 6. Condiciones de entrega

- La tarea se realizará individualmente (esto es grupos de una persona), sin excepciones.
- La entrega debe realizarse vía `http://aula.usm.cl` en un tarball en el área designada al efecto, en el formato `tarea-2 y 3-rol.tar.gz` (`rol` con dígito verificador y sin guión).

  Dicho tarball debe contener las fuentes en LaTeX 2$\varepsilon$ (al menos `tarea-2 y 3.tex`) de la parte escrita de su entrega, además de un archivo `tarea-2 y 3.pdf`, correspondiente a la compilación de esas fuentes.
- Si se utiliza algún código, idea, o contenido extraído de otra fuente, este **debe** ser citado en el lugar exacto donde se utilice, en lugar de mencionarlo al final del informe.
- Asegúrese que todas sus entregas tengan sus datos completos: número de la tarea, ramo, semestre, nombre y rol. Puede incluirlas como comentarios en sus fuentes LaTeX (en TeX comentarios son desde % hasta el final de la línea) o en posibles programas. Anótese como autor de los textos.
- Si usa material adicional al discutido en clases, detállelo. Agregue información suficiente para ubicar ese material (en caso de no tratarse de discusiones con compañeros de curso u otras personas).
- No modifique `preamble.tex`, `tarea_main.tex`, `condiciones.tex`, estructura de directorios, nombres de archivos, configuración del documento, etc. Sólo agregue texto, imágenes, tablas, código, etc. En el códigos funte de su informe, no agregue paquetes, ni archivos .tex (a excepción de que agregue archivos en `/tikz`, donde puede agregar archivos .tex con las fuentes de gráficos en `TikZ`).
- La fecha límite de entrega es el día 10 de noviembre de 2024.

**NO SE ACEPTARÁN TAREAS FUERA DE PLAZO.**

- Nos reservamos el derecho de llamar a interrogación sobre algunas de las tareas entregadas. En tal caso, la nota de la tarea será la obtenida en la interrogación.

**NO PRESENTARSE A UN LLAMADO A INTERROGACIÓN SIN JUSTIFICACIÓN PREVIA SIGNIFICA AUTOMÁTICAMENTE NOTA 0.**