# Trabajo Práctico Final

# Teoría de Algoritmos - 1c 2025

# La 12 devs

Integrantes:

Franco Bustos 110759

Camila Aleksandra Kotelchuk 110289

Lorenzo Kwiatkowski 110239

Bruno Pezman 110457

Joaquin Miranda Iglesias 97500

Fecha: 20/07/25

Lenguaje: Python

Entrega: 1

Índice:

# Parte 1: Personal para proyectos

Expresar y explicar el problema como un problema de programación lineal.

Proponemos resolver este problema de maximización con restricciones lineales mediante el algoritmo de Simplex. Primero lo formulamos como un modelo de programación lineal identificando sus variables de decisión, su función objetivo y sus restricciones. Transformamos estos datos a la forma Slack.

N = 3 variables de decisión las cuales representan nuestro plano en la programación lineal.

x1: 1 I; 3 D --> $200MM

x2: 1 I; 2 D --> $100MM

x3: 3 I; 2 D --> $300MM

x1, x2, x3  >= 0

Max Z = 200x1 + 100x2 + 300x3

Disponemos de 17 ingenieros, 11 diseñadores. Restricciones:

M = 2 restricciones
 x1 + x2 + 3x3 <= 17 (ingenieros)

3x1 + 2x2 + 2x3 <= 11 (diseñadores)

Simplex:

- ➔ 3 variables de decisión (o básicas): x1, x2, x3
- ➔ 2 variables libres: x4, x5

Explicar como resolver el problema utilizando Simplex. Brindar pseudocódigo y explicación de su propuesta. Expresar como se representa la solución.

Para la solución mediante simplex, se presentan los datos en un espacio euclidiano de dimensión igual a la cantidad de variables de decisión. En este se establecen dichas variables y sus restricciones, lo que, en caso de existir, generará un politopo. Por definición de Simplex, en busca de maximizar o minimizar la función objetivo sin exceder ninguna restricción se visita cada vértice de dicho politopo. En cada iteración se elige una variable cuya inclusión permite mejorar el valor de la función objetivo, y se realiza un cambio entre esta y una variable saliente que se vuelve nula para mantener la factibilidad del sistema.

Este proceso continúa hasta que no existan más variables candidatas a ingresar, en ese momento podemos dar por finalizado el proceso y garantizar que se ha alcanzado una solución óptima ya que el valor de la función objetivo no puede mejorar más.

**Pseudocódigo**

Representar el problema en forma de Slack agrego x4 y x5 como variables de holgura

Seleccionar la solución inicial como solución básica (x1,x2,x3,x4,x5) = (0,0,0,17,11), Z = 0

Mientras exista un coeficiente positivo en la función objetivo:

Seleccionar una variable de entrada *ve*.

x1 (coef. 200) en nuestra primera iteración y x3 (coef. 500/3) en la segunda.

Determinar la variable de salida *vs* (aquella que se ajusta al aumentar *ve*).

x5 (11/3 < 17) en nuestra primera iteración x1 (11/2 < 40/7) en la segunda.

Realizar el proceso de pivot entre *ve* y *vs*.

x1 <-> x5 en la primera iteración, x3 <-> x1 en la segunda

Actualizar el sistema de funciones y la función objetivo.

2200/3 en la primera iteración, 1650 en la segunda

Retorno:

valores óptimos de las variables de decisión (x1, x2, x3) = (0, 0, 5.5)

valor de la función objetivo Z = 1650

## Analizar la complejidad temporal y espacial de cada uno de los pasos de su solución.

Sean n las variables de decisión y m las variables de holgura. Vamos por pasos:

- Transformamos el problema a la forma Slack: Agregamos una variable de holgura por cada restricción y convertimos cada restricción en una igualdad.
  - o complejidades: O(*n*m)* temporal y O(*n+m*) espacial
- Inicializar la solución básica: se evalúa cada restricción y se almacenan todas las variables
  - complejidades: O(*m*) temporal y O(n+m) espacial
- Selección de variable de entrada: búsqueda de coeficiente positivo en la función objetivo
  - complejidades: O(*n*) temporal y O(1) espacial.
- Selección de variable de salida: se recorren todas las restricciones para hallar la que primero se ajusta.
  - complejidades: O(*m*) temporal y O(1) espacial.
- Pívot: se despeja la variable saliente y se la reemplaza en la función objetivo y las restricciones.
  - complejidades: O(n*m) temporal y O(n*m) espacial (tabla original).
- Iteraciones: la máxima cantidad de iteraciones del algoritmo de Simplex está acotada por la cantidad de vértices extremos del politopo
  - $\binom{n+m}{m}$
- TOTAL:
  - Con k cantidad máxima de iteraciones y k = $\binom{n+m}{m}$
  - La complejidad temporal total resulta O(*k*n*m*). Puede resultar exponencial con una gran cantidad de vértices extremos.
  - La complejidad espacial total resulta O(*n*m*).

Presente paso a paso la solución del problema. Presente esquemáticamente cada iteración del algoritmo. Indique qué decisiones se toma en cada uno de estos y los cálculos realizado

Primero presentamos las variables y sus restricciones, así como la composición de la función objetivo que en este caso buscamos maximizar:

x1, x2, x3 >= 0

x1 + x2 + 3x3 <= 17

3x1 + 2x2 + 2x3 <= 11

Z = 200x1 + 100x2 + 300x3

Simplex:

3 variables de decisión (o básicas): x1, x2, x3

2 variables libres: x4, x5

Expresamos la solución como (x1, x2, x3, x4, x5)

la solución básica es (0, 0, 0, 17, 11)

recorro los extremos del politopo aumentando lo máximo posible el valor objetivo

Primera iteración: selecciono una variable básica y aumento lo máximo posible su valor respetando restricciones.

(x4): 0 = 17 – x1 – 0x2 – 0x3 --> x1 = 17

(x5): 0 = 11 – 3x1 – 0x2 – 0x3 --> x1 = 11/3

la mínima entre estas (respetará ambas restricciones) es 11/3.

esta solución corresponde a (11/3,0,0,14,2) y valor objetivo = 200*11/3 + 100*0 + 300*0 = 2200/3

x5 = 11 – 3x1 – 2x2 – 2x3 --> x1 = 11/3 – 2/3x2 – 2/3x3 – x5/3

La variable entrante en esta iteración es x1 y la saliente x5

Segunda iteración: busco máx Z = 2200/3 – 100/3x2 + 500/3x3

tengo x1 y x4 variables básicas, x2 = 0 y x5 = 0 (no están en la base) y busco aumentar x3

Selecciono la variable con coeficiente positivo y nuevamente busco su valor máximo entre las nuevas restricciones:

con x1 = 11/3 – 2/3x2 – 2/3x3 y x4 = 17 – x1 – x2 – 3x3 reemplazo en x4:

x4 = 17 – (11/3 – 2/3x2 – 2/3x3) – x2 – 3x3 --> x4 = 40/3 – 1/3x2 – 7/3x3 --> x4 >= 0 -->

1/3x2 – 7/3x3 <= 40/3 entonces busco cómo aumenta x3 respetando las restricciones:

(x1): 0 = 11/3 – 2/3*0 – 2/3x3 --> x3 = 11/2 = 5.5

(x4): 0 = 40/3 – 7/3x3 --> x3 = 40/7 ~ 5.7

la mínima entre las dos es 5.5, la cual maximiza x3 sin llevar a ninguna variable básica a un valor negativo. Aumentamos x3 a 5.5 y x1 queda en 0. En este caso x3 es la variable entrante y x1 la variable saliente.

La nueva solución se expresa como (0, 0, 5.5, 0.5, 0) y en la función objetivo queda:

Z = 200x1 + 100x2 + 300x3 = 300 · 5.5 = 1650. Con 1650 > 2200/3 ~ 733.3

Tercera iteración:

Con x1 = 11/3 – 2/3x2 – 2/3x3; x4 = 40/3 – 1/3x2 – 7/3x3 y Z = 2200/3 – 100/3x2 + 500/3x3.

Escribo x3 desde la función de x1:

x1 = 11/3 – 2/3x2 – 2/3x3 --> x3 = 11/2 – x2 – 3/2x1

Reemplazo x3 en x4:

x4 = 40/3 – 1/3x2 – 7/3x3 = 40/3 – 1/3x2 – 7/3 (11/2 – x2 – 3/2x1) = 40/3 – 1/3x2 – (77/6 – 7/3x2 – 7/2x1) = 40/3 – 1/3x2 – 77/6 + 7/3x2 + 7/2x1 -->

x4 = 1/2 + 2x2 + 7/2x1

Reemplazo en Z (x3) en función de (x1):

Z = 2200/3 – 100/3x2 + 500/3x3 = 2200/3 – 100/3x2 + 500/3(11/2 – x2 – 3/2x1) =

Z = 1650 – 200x2 – 125x1

¿Cuarta iteración?

Como no existen variables con coeficientes positivos en la función objetivo, se ha alcanzado la solución óptima según el algoritmo Simplex:

Valor = 1650

Solución: (x1, x2, x3) = (0, 0, 5.5)

Analizar la solución obtenida. ¿Corresponde a la solución óptima del problema? Según la naturaleza del problema, ¿lo puede encontrar en tiempo polinomial? Justificar.

La solución obtenida fue (0,0,5.5) que consiguió un valor de 1650. Esta solución cumple con todas las restricciones del problema y llega al valor máximo de Z sin violar ninguna condición impuesta por los recursos.

En cuanto a la complejidad, si bien el algoritmo Simplex no garantiza su ejecución en tiempo polinomial en todos los casos, en este problema sí se ejecutó en tiempo polinomial. En este caso se llegó a la solución óptima en tan solo 2 iteraciones. En el peor caso por la pequeña cantidad de variables de decisión (n=3) y restricciones (m=2) la solución aún se alcanzaría en tiempo polinomial ya que $\binom{n+m}{m}$ es la cota superior de la cantidad de iteraciones y con nuestros valores es igual a 10 por lo que la complejidad se puede reducir a O(n*m) es decir, tiempo polinomial.

Investigar la aplicación de Branch and Bound dentro de programación lineal entera. Explique cómo funciona paso a paso. Brinde pseudocódigo. ¿Lo puede utilizar en el problema? En caso afirmativo desarrolle como hacerlo paso a paso.

La solución óptima obtenida mediante programación lineal asume que las variables de decisión pueden tomar valores reales. Sin embargo, en la práctica, los proyectos y los recursos humanos involucrados no son fraccionables y en estos casos se puede reformar el problema como uno de programación lineal entera. Se puede utilizar branch and bound con este fin.

Como Simplex no garantiza soluciones enteras, B&B extiende Simplex generando un árbol de decisión donde en cada rama se imponen condiciones que fuerzan soluciones enteras. El método consiste en primero resolver el problema mediante Simplex normalmente. Luego se analiza si alguna variable no es entera, si todas son enteras listo, encontramos la solución óptima y entera. Si alguna variable no es entera se ramifica el problema.

En la ramificación (justamente “branching”) se elije una variable xi con valor no entero y se crean dos subproblemas, uno con xi <= a y otro con xi >= b con: “a” el valor original redondeado a su valor entero para abajo, y “b” con el valor original redondeado a su valor entero para arriba. Cada subproblema se resuelve nuevamente con Simplex y se filtran las soluciones:

- ➔ si viola una restricción se la descarta (inviabilidad)
- ➔ Si no es entera, pero consigue peor valor que una solución ya encontrada, se la descarta (poda)
- ➔ si es entera se la guarda como potencial candidata

Esta ramificación se repite con todos los nodos con soluciones no enteras y potencial para mejorar y al final se elige la mejor solución entera factible.

En caso de nuestro problema sí se puede utilizar y es lo más lógico dado que los recursos que se presentan como restrictivos son no divisibles. En el caso de nuestro problema la solución sería así:

Empezamos tomando el resultado de nuestra primera solución por Simplex y analizamos si hay valores no enteros. {x1 = 0; x2 = 0; x3 = 5.5; Z = 1650} con x3 no entero. Aplicamos “branching” y nos queda el nodo izquierdo x3 <= 5 y el nodo derecho x3 >= 6. Resolvemos cada nodo con Simplex. x3 <= 5 respeta las restricciones y da como solución {x1 = 0; x2 = 0; x3 = 5; Z = 1500} por lo que la guardamos como una solución factible. x3 >= 6 viola las restricciones por lo que se la descarta por inviabilidad. No hay más valores fraccionarios por lo que conservamos la mejor solución entera y factible. Obtuvimos como resultado de plantear el problema como PLE la solución {x1 = 0; x2 = 0; x3 = 5; Z = 1500}.

Programe su propuesta. Puede utilizar librerías para Simplex. (Ejemplos: ALGLIB o PulP).

En nuestra solución utilizamos funciones de la librería “PuLP” la cual es necesario instalar (se instala ingresando por consola “$ pip install pulp”. Los datos de los recursos y proyectos los recibimos en un archivo “.txt” con un formato

```
recursos: x1={int}, x2={int}, ..., xn={int}

proyectos:

xA: x1={int}, x2={int}, ganancia={int}
```

```
xB: x1={int}, x2={int}, ganancia={int}

...

xC: x1={int}, x2={int}, ganancia={int}
```

y se corre ingresando por consola:

```
$ python recursos_proyectos_ple.py <ruta archivo.txt>
```

# Parte 2: Buscando una solución aceptable.

## Analizar el problema y determinar si es posible resolverlo de forma eficiente.

Tenemos:

- ➔ Conjunto de inversores $I = \{I_1, I_2, ..., I_n\}$
- ➔ Cada inversor $I_i$ aporta un monto $W_j > 0$
- ➔ Algunas parejas de inversores son incompatibles, es decir, no pueden ser seleccionados conjuntamente.

El objetivo es determinar un subconjunto de inversores compatibles entre sí, de forma tal que la suma del dinero aportado sea máxima.
Este problema puede modelarse como un grafo donde cada nodo representa un inversor, hay una arista entre dos nodos si los inversores son incompatibles y el peso de cada nodo es el monto de dinero que ese inversor ofrece. Entonces, el problema equivale a encontrar un conjunto independiente de vértices (ninguna arista entre ellos), de peso total máximo. Este problema es una generalización del Maximum Independent Set, que es NP-hard. La versión con pesos también es NP-hard.
Por tanto, no se puede resolver eficientemente en el caso general (es decir, no se conoce ningún algoritmo polinomial para resolverlo en todos los casos, y es improbable que exista).

## Proponer 2 algoritmos heurísticos/randomizados o de aproximación para resolverlo. Explicar cómo funcionan

**Simulated Annealing**

Este algoritmo heurístico se basa en una estrategia de búsqueda local con aceptación probabilística de soluciones peores, lo que le permite escapar de óptimos locales. A diferencia de algoritmos greedy tradicionales, Simulated Annealing puede aceptar soluciones de menor calidad en ciertos momentos, con una probabilidad que disminuye progresivamente según un esquema de enfriamiento.

Consideramos adecuado aplicar este enfoque a nuestro problema, ya que nos permite explorar diversas configuraciones de inversores, manteniendo siempre la mejor solución factible encontrada hasta el momento, independientemente del estado actual. De este modo, se promueve una exploración más amplia del espacio de soluciones.

Para avanzar desde un estado inicial, definimos una función de vecindad que genera un estado vecino (es decir, otra selección de inversores compatibles) de manera aleatoria. Si el nuevo estado tiene un valor de función objetivo superior (mayor inversión total), se aceptará. En caso contrario, se acepta con una cierta probabilidad que depende de la temperatura y de la función costo de los estados vecino y actual. Esta comienza con un valor inicial $T_0 > 0$ y va descendiendo dado que se multiplica a T por una constante C < 1 luego de cada iteración.

**Tabu Search**

La incompatibilidad entre ciertos pares de inversores hace especialmente adecuado el uso de Tabu Search para este problema dado que permite explorar el espacio de soluciones evitando ciclos o regresos a configuraciones previamente visitadas, mediante el uso de una lista tabú que restringe ciertos movimientos.

En este contexto, podemos considerar como movimientos tabú aquellos que implican incluir simultáneamente a inversores incompatibles, es decir, transiciones que violan las restricciones del problema. Estos movimientos se almacenan temporalmente en la lista tabú, impidiendo que el algoritmo reincida en configuraciones inviables o poco prometedoras.

Sin embargo, bajo ciertas condiciones, llamadas criterios de aspiración, es posible anular la prohibición de un movimiento tabú si conduce a una solución que mejora sustancialmente la mejor encontrada hasta el momento. Utilizaremos mecanismos llamados intensificación o diversificación que impactaran en la función costo. De esta manera, nuestro algoritmo aprende de movimientos útiles (intensificación) y evita caer en rutinas o bucles (diversificación). Se contará cada movimiento seleccionado y un indicador relativo a los incrementos o decrementos logrados en su elección. Estos valores normalizados se aplican a la FC. Utilizaremos constantes $\alpha$ y $\beta$ para controlar explícitamente el peso que le damos a la intensificación y diversificación. Se tomó esta decisión porque de no hacerse, estaríamos asumiendo implícitamente que los valores normalizados están en una escala comparable. Además, tal vez la diversificación termina castigando demasiado o la intensificación no influye lo suficiente.

En ambos algoritmos se genera el próximo estado vecino con los movimientos de agregar, quitar o intercambiar inversores de un arreglo solución, aprovechando la aleatoriedad permite que no haya sesgo hacia un solo tipo de modificación y que el algoritmo tenga mayor libertad para moverse.

- ➔ AGREGAR tiende a diversificar (nuevos caminos).
- ➔ INTERCAMBIAR y QUITAR tienden a intensificar (mejorar en una misma región).
- ➔ Al tener las tres, el algoritmo naturalmente equilibra estos dos procesos sin que tengas que forzarlo manualmente.

Brindar pseudocódigo y análisis completo del mismo (análisis de complejidad temporal, espacial, optimalidad, grado de aproximación, según corresponda al tipo de solución).

**Observación:** Para los dos algoritmos utilizaremos el código que genera un estado solución factible inicial y el código que chequea compatibilidad.

**Simulated Annealing**
Sea I el conjunto de inversores preseleccionados $\{i_1, i_2, ..., i\square\}$
Sea W(i) el monto prometido por cada inversor $i \in I$
Sea Incompatibles[i] contiene todos los inversores con los que el inversor i no es compatible
Sea C la constante de enfriamiento (0 < C < 1)
Sea MAX la cantidad máxima de iteraciones

Sea IT=0 la cantidad de iteraciones actuales
Sea Sm el estado de función máxima encontrado hasta el momento
Sea S el estado actual del problema
Sea $S_0$ el estado inicial generado como subconjunto inicial de inversores compatibles
Sea $T_0$ la temperatura inicial
Establecer $S=S_0$ y $Sm = S_0$ como estado inicial

Mientras IT < MAX
    Seleccionar de forma aleatoria Sv un estado vecino de S a partir de un intercambio con otro inversor compatible
    Sea siguiente estado actual dado por determinar_el_siguiente_estado_actual( T, S, Sv)
    Establecer Sm como el estado de costo máximo entre Sm y S
    Realizar enfriamiento T= T*C
    Incrementar IT en 1

Retornar Sm

generar_estado_inicial():
    S = ∅
    I_aleatorio = aleatorizar(I)

    Para cada inversor i en I_aleatorio:
        Si es_factible(S ∪ {i}):
            Agregar i a S
    Retornar S


es_factible(S):
    Para cada inversor i en S hacer:
        Para cada j en incompatibles[i] hacer:
            Si j ∈ S entonces:
                retornar false
    retornar true

determinar_el_siguiente_estado_actual( T, S, Sv):
    Sea C calculada por suma_de_montos(S)
    Sea Cv calculada por suma_de_montos(Sv)

    Si Cv > C
        retornar SV

    Calcular R de forma aleatoria entre 0 y 1
    Calcular exp = (Cv - C) / T
    Si R < (e elevado a exp)
        retornar Sv
    Retornar S

suma_de_montos(S):
    total = 0
    para cada inversor i en S:
        total = total + W(i)
    retornar total

```
generar_vecino(S):
    Mientras es_factible(S) diga no:
        Sea S' una copia de S
        Sea OP una operación aleatoria entre {agregar, quitar, intercambiar}

        Si OP es agregar
            Seleccionar inversor i ∈ I − S al azar
            Agregar i a S'
        Si OP es quitar
            Si S no está vacío:
                Seleccionar inversor i ∈ S al azar
                Quitar i de S'
        Si OP es intercambiar
            Si S no está vacío:
                Seleccionar i ∈ S al azar
                Seleccionar j ∈ I − S al azar
                Quitar i de S' y agregar j

    Retornar S'
```

Complejidad temporal**:** Sea n cantidad de inversores y m cantidad de pares incompatibles

- ➔ O(n+m): Subconjunto inicial de inversores compatibles
- ➔ O(n + m): Chequear incompatibilidad en el peor caso
- ➔ O(n): Calcular el valor de la FC
- ➔ O(n + m): Generar un estado vecino compatible
- ➔ O(1): Cálculos probabilísticos
- ➔ O(MAX*(n + m)): total con iteración teniendo en cuenta lo anterior
- ➔ TOTAL: O(n + m)

Complejidad espacial**:** considerando que ya tenemos la lista de inversores, sus respectivas inversiones y una matriz de incompatibles.

- ➔ O(n): Guardar los 3 estados Sv, S, Sm si todos los inversores son compatibles
- ➔ O(1): Variables auxiliares
- ➔ TOTAL: O(n)

**Tabu Search**

Sea I el conjunto de inversores preseleccionados {$i_1$, $i_2$, ..., i□}
Sea W(i) el monto prometido por cada inversor i ∈ I
Sea Incompatibles[i] contiene todos los inversores con los que el inversor i no es compatible
Sea MAX la cantidad máxima de iteraciones
Sea iter_actual = 0 la cantidad de iteraciones actuales
Sea S el estado inicial generado como subconjunto inicial de inversores compatibles
Sea DicTabu el diccionario movimiento → cant_iteraciones
Sea freq[m] el diccionario movimiento → entero
Sea delta_acumulado[m] el diccionario movimiento → float
Definimos $\alpha$ = 0.5
Definimos $\beta$ = 0.3
Establecer Sm = S
Establecer C dado suma_de_montos(S) como valor actual de FC

Mientras iter_actual < MAX

```
        Sea Sv el estado vecino de S dado por buscar_mejor_vecino(S, Sm, DicTabu, iter_actual,
        freq, delta_acumulado)
        Si Cv > C y Sv es una solución factible
                        Definir Sm como Sv.
                        Definir C como Cv.
        Definir S como Sv
        Para cada m en lista(DicTabu.keys()):
                Si iter_actual ≥ DicTabu[m]:
                        eliminar DicTabu[m]
        iter_actual++
Retornar Sm

buscar_mejor_vecino(S, Sm, DicTabu, iter_actual, freq, delta_acumulado):
        mejor_vecino = None
        mejor_score = -∞

        movimientos_posibles = ∅

        // Generar movimientos posibles: agregar
        Para i ∈ I − S:
                movimientos_posibles.agregar(("agregar", i))

         // quitar
         Para i ∈ S:
                movimientos_posibles.agregar(("quitar", i))

        Para cada movimiento m ∈ movimientos_posibles:
                S' = copia de S

                Si m = ("agregar", i):
                S'.agregar(i)
                Si m = ("quitar", i):
                S'.quitar(i)
                Si m = ("intercambiar", i, j):
                S'.quitar(i)
                S'.agregar(j)

                Si no es_factible(S'):
                continuar

                Cv = suma_de_montos(S')
                Cs = suma_de_montos(S)

                // Verificar si m está en la lista tabú
                Si m ∈ DicTabu y iter_actual < DicTabu[m]:
                        continuar  // prohibido y no cumple aspiración

                // Criterio de aspiración
                Si prohibido y Cv ≤ suma_de_montos(Sm):
                continuar

                // Normalización
```

```
                max_freq = max(freq.values() ∪ {1})
                max_delta = max({|d| for d in delta_acumulado.values()} ∪ {1})

                intensificación = delta_acumulado[m] / max_delta
                diversificación = freq[m] / max_freq

                score = Cv + α * intensificación - β * diversificación

                Si score > mejor_score:
                mejor_score = score
                mejor_vecino = (S', m, Cv - Cs)

        Si mejor_vecino ≠ None:
                (Sv, m, mejora) = mejor_vecino
                freq[m] += 1
                delta_acumulado[m] += mejora
                DicTabu[m] = iter_actual + IT_TABU
                retornar Sv
        Sino:
                retornar S
```

Complejidad Temporal**:** Sea n cantidad de inversores y m cantidad de pares incompatibles. Por cada iteración determinada por MAX. En este caso, la complejidad está determinada por la búsqueda del estado vecino dado que el bucle principal ejecuta MAX iteraciones y en cada una:

1. Llama a buscar_mejor_vecino(S, ...)
2. Compara con la mejor solución y actualiza
3. Limpia DicTabu

Dentro de buscar_mejor_vecino:

- ➔ O(n): Generación de movimientos posibles
- ➔ Por cada movimiento:
  - ◆ O(n): Copiar el estado
  - ◆ O(1): Modificar el estado
  - ◆ O(n + m): es_factible(S)
  - ◆ O(n): Suma de montos
  - ◆ O(1): Ajustar el score por intensificación/diversificación
  - ◆ Total: O(n + m)
- ➔ Total con iteraciónes: O(MAX($n^2$ +n*m))
- ➔ TOTAL: O($n^2$ +n*m)

Complejidad Espacial: considerando que ya tenemos la lista de inversores, sus respectivas inversiones y una de incompatibles.

- ➔ O(n): Guardar los 3 estados Sv, S, Sm en el peor caso
- ➔ O(n): freq, delta_acumulado
- ➔ O(n): DicTabu
- ➔ TOTAL: O(n)

## Realizar su programa y brindar varios ejemplos para su ejecución

Para probar los algoritmos de **Simulated annealing** y **Tabu Search** se deben correr de la siguiente manera:

- **Simulated annealing:**

  *python sa.py inversores1.txt montos1.txt incompatibles1.txt (ejemplo 1)*

  *python sa.py inversores2.txt montos2.txt incompatibles2.txt (ejemplo 2)*

  *python sa.py inversores3.txt montos3.txt incompatibles3.txt (ejemplo 3)*

- **Tabu Search:**

  *python ts.py inversores1.txt montos1.txt incompatibles1.txt (ejemplo 1)*

  *python ts.py inversores2.txt montos2.txt incompatibles2.txt (ejemplo 2)*

  *python ts.py inversores3.txt montos3.txt incompatibles3.txt (ejemplo 3)*

*aclaración: el programa "utils_inversores.py" es un archivo auxiliar que tiene el desarrollo de las funciones para leer la entrada estandar y las que comparten entre si ambos algoritmos. El mismo es usado tanto en "ts.py" y "sa.py"*

En el ejemplo 1, buscaremos testear la estabilidad y precisión de ambos algoritmos en un caso conflictivo.

| **Inversores** | **Montos** | **Incompatibilidades** |
|---|---|---|
| A | 5 | B,C,D |
| B | 8 | A,E |
| C | 3 | A |
| D | 9 | A,F |
| E | 4 | B |
| F | 6 | D |

En el ejemplo 2, veremos qué algoritmo explora mejor un espacio más grande, pero con más combinaciones válidas.

| **Inversores** | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Montos** | 10 | 15 | 8 | 7 | 12 | 9 | 11 | 14 | 13 | 6 | 5 | 10 |
| **Incompatibles** | B | A | D | C | F | E | H | G | J | I | L | K |

En el ejemplo 3, probaremos la escalabilidad y robustez de los algoritmos

| **Inversores** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Montos** | 88 | 84 | 89 | 89 | 74 | 43 | 94 | 42 | 88 | 19 | 40 | 84 | 33 | 75 | 24 |
| **Incompatibilidades** | B,D,Q,U,W,Z | A,G,J,P | O,Y | A,T | M,AA | J,R,AD | B,H,AC | G | M | B,F | V | X | E,I | T | C |

| **Inversores** | P | Q | R | S | T | U | V | W | X | Y | Z | AA | AB | AC | AD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Montos** | 77 | 90 | 72 | 56 | 71 | 23 | 81 | 88 | 64 | 35 | 74 | 98 | 15 | 32 | 80 |
| **Incompatibilidades** | B | A | F | AB | D,N | A,AD | K | A | L | C | A | E | S | G | F,U |

Realizar un análisis empírico de sus soluciones. ¿Puede determinar en qué casos es mejor uno que el otro?

Realizamos 5 ejecuciones independientes para cada ejemplo utilizando ambos algoritmos: Tabu Search (TS) y Simulated Annealing (SA). Adoptamos así un enfoque de multi-start, con el objetivo de evaluar no solo la calidad de las soluciones encontradas, sino también la estabilidad, consistencia y robustez de cada algoritmo frente a distintas condiciones iniciales.

*Análisis del resultado con ejemplo 1*

| **Ejecuciones** | **Simulated Annealing** | **Tabu Search** |
|---|---|---|
| 1 | {C,B,D} Monto: 20 | {D,B,C} Monto: 20 |
| 2 | {B,D} Monto: 17 | {D,B,C} Monto: 20 |
| 3 | {E,F,A} Monto: 15 | {D,B,C} Monto: 20 |
| 4 | {E,F,A} Monto: 15 | {D,B,C} Monto: 20 |
| 5 | {C,B,D} Monto: 20 | {D,C,B} Monto: 20 |

En esta instancia con pocos inversores y un alto grado de incompatibilidad, TS demostró ser más estable y confiable. Encontró la solución de máxima ganancia con mayor frecuencia que SA y además mantuvo más consistencia en la composición del conjunto de inversores seleccionados.

*Análisis ejemplo 2*

| **Ejecuciones** | **Simulated Annealing** | **Tabu Search** |
|---|---|---|
| 1 | {E,H,C,I,B,D} Monto: 71 | {B,L,I,C,H,E} Monto: 72 |
| 2 | {K,F,A,J,H,C} Monto: 52 | {B,L,I,H,C,E} Monto: 72 |
| 3 | {K,F,H,C,B,I} Monto: 64 | {B,L,C,I,H,E} Monto: 72 |
| 4 | {J,A,C,L,G,E} Monto: 57 | {B,L,I,H,C,E} Monto: 72 |
| 5 | {J,B,H,C,L,E} Monto: 65 | {B,L,I,C,H,E} Monto: 72 |

En esta instancia con una cantidad intermedia de inversores y un número relativamente bajo de incompatibilidades, TS volvió a destacar frente a SA. La mayor cantidad de combinaciones válidas

favorece a ambos algoritmos, pero TS mostró una mejor escalabilidad y mayor consistencia en los resultados. Las diferencias entre ejecuciones de TS fueron mínimas (a lo sumo en el orden de selección), mientras que SA presentó una alta dispersión en las ganancias obtenidas, evidenciando su sensibilidad a parámetros y aleatoriedad. Esto sugiere que TS es más robusto cuando el espacio de búsqueda es amplio.

*Análisis ejemplo 3*

| **Ejecuciones** | **Simulated Annealing** | **Tabu Search** |
|---|---|---|
| 1 | {'J', 'R', 'X', 'T', 'I', 'Q', 'H', 'AC', 'P', 'Z', 'V', 'AB', 'W', 'E', 'Y', 'O', 'AD'}<br><br>Monto: 1026 | {'Z', 'J', 'AD', 'D', 'AA', 'Y', 'S', 'N', 'L', 'W', 'H', 'Q', 'O', 'I', 'V', 'P', 'AC', 'R'}<br><br>Monto: 1204 |
| 2 | {'T', 'R', 'X', 'Q', 'B', 'H', 'AC', 'Z', 'AB', 'W', 'Y', 'AA', 'M', 'O', 'K', 'U'}<br><br>Monto: 885 | {'Z', 'D', 'AD', 'AA', 'J', 'Y', 'N', 'S', 'L', 'W', 'H', 'Q', 'O', 'I', 'V', 'P', 'AC', 'R'}<br><br>Monto: 1204 |
| 3 | {'J', 'R', 'T', 'X', 'I', 'Q', 'H', 'AC', 'P', 'Z', 'S', 'V', 'W', 'Y', 'AA', 'O', 'AD'}<br><br>Monto: 1091 | {'Z', 'D', 'AD', 'AA', 'J', 'N', 'S', 'L', 'W', 'H', 'Q', 'I', 'V', 'P', 'AC', 'R', 'C'}<br><br>Monto: 1234 |
| 4 | {'D', 'J', 'R', 'I', 'Q', 'P', 'G', 'N', 'Z', 'S', 'W', 'Y', 'AA', 'O', 'K', 'L', 'U'}<br><br>Monto: 1126 | {'Z', 'D', 'AA', 'AD', 'J', 'S', 'N', 'L', 'W', 'H', 'Q', 'I', 'V', 'P', 'AC', 'R', 'C'}<br><br>Monto: 1234 |
| 5 | {{'D', 'I', 'X', 'Q', 'F', 'P', 'G', 'Z', 'N', 'V', 'AB', 'W', 'Y', 'AA', 'O', 'U'}<br><br>Monto: 1058 | {'Z', 'AA', 'AD', 'J', 'D', 'S', 'N', 'G', 'L', 'W', 'Q', 'I', 'V', 'P', 'R', 'C'}<br><br>Monto: 1254 |

En esta instancia con 30 inversores y un nivel considerable de incompatibilidades, el comportamiento de ambos algoritmos muestra diferencias más marcadas. La variabilidad de los resultados es mayor en ambos, pero especialmente en SA, donde se observan diferencias notorias entre ejecuciones, como en la segunda ejecución, que alcanzó un valor máximo de 885. Esto sugiere que SA pudo haberse quedado atrapado en un óptimo local del cual no pudo salir. En contraste, TS ofreció un rendimiento más estable y, en promedio, logró mejores soluciones globales. Si bien también se observa variabilidad, esta es mucho menor que en SA. Esto evidencia que TS maneja mejor la escalabilidad y la navegación del espacio de búsqueda en problemas grandes y con alta complejidad estructural.

*Conclusión*

El algoritmo Tabu Search se mostró superior a Simulated Annealing para resolver el problema de selección de inversores compatibles con máxima ganancia. Esta diferencia se debe, en gran medida, al uso de una lista de movimientos tabú, que permite evitar ciclos y escapar más efectivamente de los óptimos locales.

A lo largo de los experimentos, Tabu Search fue capaz de encontrar soluciones globales de mayor calidad con una estabilidad notable entre ejecuciones, incluso en instancias grandes y conflictivas. Aunque presenta una complejidad temporal mayor por el mantenimiento de la lista tabú y la búsqueda de vecinos más intensiva, esto se ve compensado por su robustez, menor variabilidad y mejor rendimiento promedio, especialmente a gran escala.

# Parte 3: Autómatas finitos

Construir un autómata finito determinista (AFD) de forma formal para cada uno de los lenguajes simples.

Basándose en los autómatas construidos generar de forma formal los dos autómatas finitos determinísticos de los lenguajes más complejos.

Representar cada uno de los AFD mediante su representación gráfica

Para cada AFD Brindar un ejemplo de una cadena que acepta y otra que rechaza. Explicar cómo determinar su autómata si es aceptado o rechazado paso a paso.

Tenemos los siguientes lenguajes:

- Lenguaje "a": {w / w (tiene dentro suyo la subcadena "1011") y (tiene un número par de "0")}
- Lenguaje "b": {w / w (toda posición impar es un "1") y (termina con 3 "1")}

El primer paso es identificar los lenguajes simples:

- Lenguaje "a": $L_a = L_{a1} \cap L_{a2}$ siendo:
  - $L_{a1}$ = {w / w tiene dentro suyo la subcadena "1011"}
  - $L_{a2}$ = {w / w tiene un número par de "0"}
- Lenguaje "b": $L_b = L_{b1} \cap L_{b2}$ siendo:
  - $L_{b1}$ = {w / w toda posición impar es un "1"}
  - $L_{b2}$ = {w / w termina con 3 "1"}

1. Construir un autómata finito determinista (AFD) de forma formal para cada uno de los lenguajes simples.

- Lenguaje simple $L_{a1}$ – contiene la subcadena "1011":
  - $L_{a1}$ = {w / w tiene dentro suyo la subcadena "1011"}
  - $M_{a1} = (Q, \Sigma, \delta, q_0, F)$, donde:
    - $Q = \{q_0, q_1, q_2, q_3, q_4\}$
    - $\Sigma = \{0, 1\}$
    - $q_0$ estado inicial
    - $F = \{q_4\}$

| | 0 | 1 |
|---|---|---|
| $q_0$ | $q_0$ | $q_1$ |
| $q_1$ | $q_2$ | $q_1$ |
| $q_2$ | $q_0$ | $q_3$ |
| $q_3$ | $q_2$ | $q_4$ |
| $q_4$ | $q_4$ | $q_4$ |

- Lenguaje simple $L_{a2}$ – tiene un número par de "0":
  - $L_{a2}$ = {w / w tiene un número par de "0"}

- $M_{a2} = (Q, \Sigma, \delta, q_0, F)$, donde:
  - $Q = \{q_0, q_1\}$
  - $\Sigma = \{0, 1\}$
  - $q_0$ estado inicial
  - $F = \{q_0\}$

| | 0 | 1 |
|---|---|---|
| $q_0$ | $q_1$ | $q_0$ |
| $q_1$ | $q_0$ | $q_1$ |

- Lenguaje simple $L_{b1}$ – toda posición impar es un "1":
  - $L_{b1}$ = {w / w toda posición impar es un "1"}
  - $M_{b1} = (Q, \Sigma, \delta, q_0, F)$, donde:
    - $Q = \{q_0, q_1, q_2\}$
    - $\Sigma = \{0, 1\}$
    - $q_0$ estado inicial
    - $F = \{q_1\}$

| | 0 | 1 |
|---|---|---|
| $q_0$ | $q_2$ | $q_1$ |
| $q_1$ | $q_0$ | $q_0$ |
| $q_2$ | $q_2$ | $q_2$ |

- Lenguaje simple $L_{b2}$ – w termina con 3 "1":
  - $L_{b2}$ = {w / w termina con 3 "1"}
  - $M_{b2} = (Q, \Sigma, \delta, q_0, F)$, donde:
    - $Q = \{q_0, q_1, q_2, q_3\}$
    - $\Sigma = \{0, 1\}$
    - $q_0$ estado inicial
    - $F = \{q_3\}$

| | 0 | 1 |
|---|---|---|
| $q_0$ | $q_0$ | $q_1$ |
| $q_1$ | $q_0$ | $q_2$ |
| $q_2$ | $q_0$ | $q_3$ |
| $q_3$ | $q_0$ | $q_3$ |

2. Basándose en los autómatas construidos generar de forma formal los dos autómatas finitos determinísticos de los lenguajes más complejos.

# -AFD de La-

## *Definiciones previas*

**AFD de La1:**

- $L_{a1}$ = {w / w tiene dentro suyo la subcadena “1011”}
- $M_{a1}$ = (Q, Σ, δ, $q_0$, F), donde:
  - Q = {$q_0$, $q_1$, $q_2$, $q_3$, $q_4$}
  - Σ = {0, 1}
  - $q_0$ estado inicial
  - F = {$q_4$}

**AFD de La2:**

- $L_{a2}$ = {w / w tiene un número par de “0”}
- $M_{a2}$ = (Q, Σ, δ, $q_0$, F), donde:
  - Q = {$q_0$, $q_1$}
  - Σ = {0, 1}
  - $q_0$ estado inicial
  - F = {$q_0$}

**Lenguaje complejo La = La1 ∩ La2**

El nuevo **AFD** se define como:

- La = {w / w contiene la subcadena "1011" y tiene un nu´mero par de “0”}
- Ma = (Q, Σ, δ, q0, F), donde:

  - **Conjunto de estados**:
    Q = Qa1×Qa2 = {(qi, pj) | qi ∈ Qa1, pj ∈ Qa2} ⇒ 5 × 2 = **10 estados**

  - **Alfabeto**:
    Σ = {0, 1}

  - **Estado inicial**:
    (q0, p0)

  - **Estados finales**:
    F = {(q1, p3)} -> (solo se acepta si **se llegó a q4** y **hay cantidad par de ceros**)

- **Función de transición**:
  $\delta((qi, pj), a) = (\delta1(qi, a), \delta2(pj, a))$

  - $\delta1$ es la función de transición del autómata para La1 (formación secuencia "1011")

  - $\delta2$ es la función de transición del autómata para La2 (cantidad par de "0")

## -AFD de Lb-

### *Definiciones previas*

**AFD de Lb1:**

- $L_{b1}$ = {w / w toda posición impar es un "1"}
- $M_{b1} = (Q, \Sigma, \delta, q_0, F)$, donde:
  - $Q = \{q_0, q_1, q_2\}$
  - $\Sigma = \{0, 1\}$
  - $q_0$ estado inicial
  - $F = \{q_1\}$

**AFD de Lb2:**

- $L_{b2}$ = {w / w termina con 3 "1"}
- $M_{b2} = (Q, \Sigma, \delta, q_0, F)$, donde:
  - $Q = \{q_0, q_1, q_2, q_3\}$
  - $\Sigma = \{0, 1\}$
  - $q_0$ estado inicial
  - $F = \{q_3\}$

**Lenguaje complejo Lb = Lb1 ∩ Lb2**

El nuevo **AFD** se define como:

- Lb={ w ∣ toda posicio´n impar es un "1" y w termina con "111"}
- Mb=(Q, Σ, δ, $q_0$, F), donde:
  - **Conjunto de estados**:
    $Q = Qa1 \times Qa2 = \{(qi, pj) \mid qi \in Qa1, pj \in Qa2\} \Rightarrow 3 \times 4 =$ **12 estados**

  - **Alfabeto**:
    $\Sigma = \{0, 1\}$

- **Estado inicial**:
  (q0, p0)

- **Estados finales**:
  F = {(q1, p3)} -> (**Se respetó la condición de 1s en posiciones impares** (terminando en q1), y **la palabra termina en "111"** (terminando en p3))

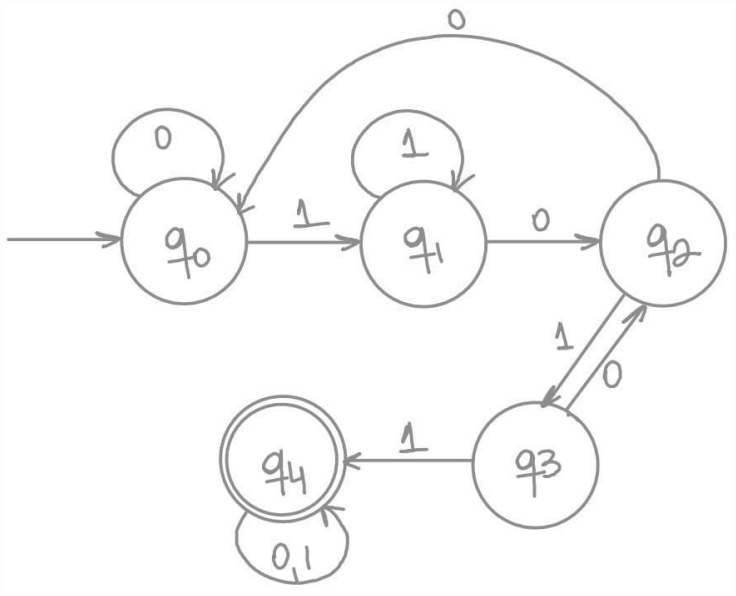
- **Función de transición**:
  δ((qi, pj), a) = (δ1(qi, a), δ2(pj, a))

  - δ1 es la función de transición del autómata para Lb1 (posición impar = 1)

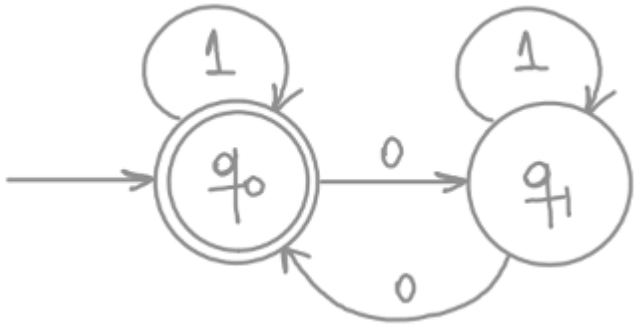
  - δ2 es la función de transición del autómata para Lb2 (termina en "111")

3. Representar cada uno de los AFD mediante su representación gráfica

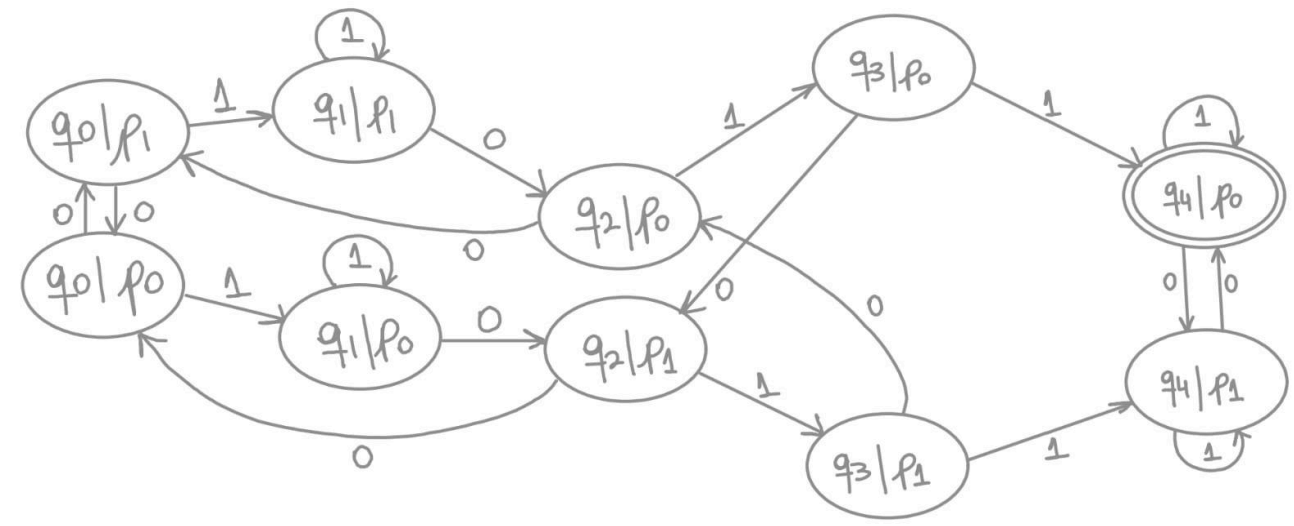
   - Lenguaje simple $L_{a1}$:



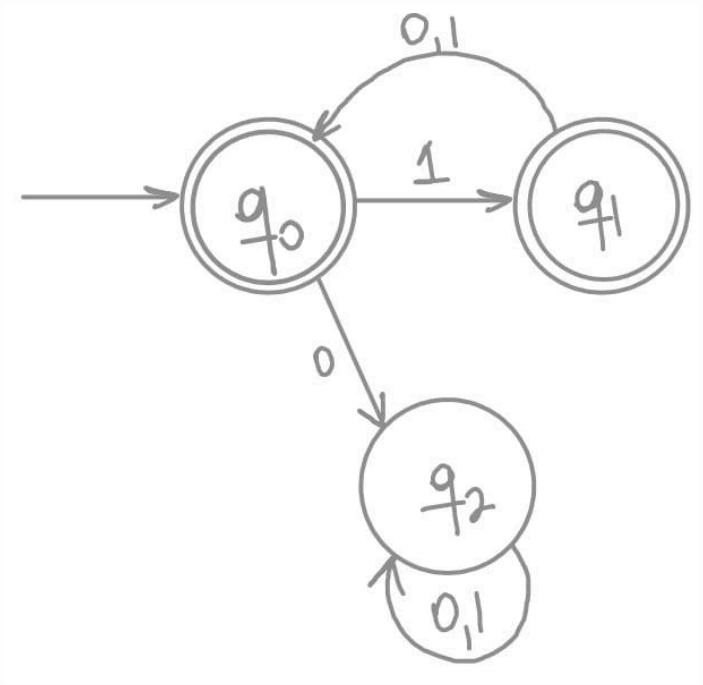
- Lenguaje simple $L_{a2}$:



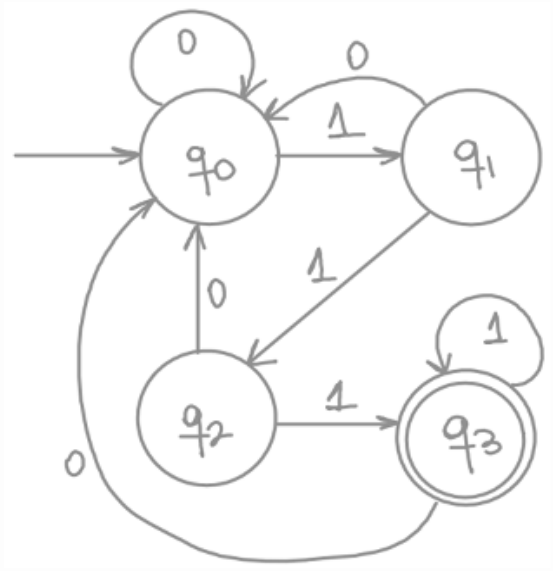

- Lenguaje complejo $L_a$:



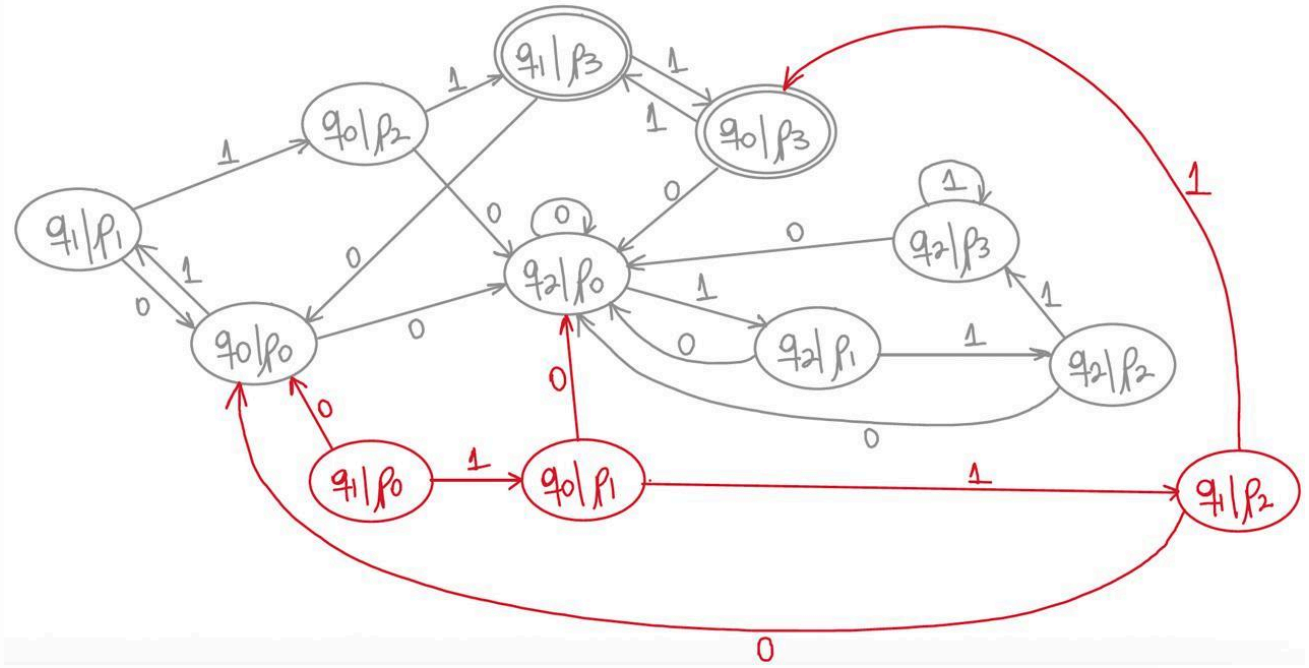
- Lenguaje simple $L_{b1}$:




- Lenguaje simple $L_{b2}$:



- Lenguaje simple $L_b$:




## Observaciones sobre el autómata b:

Al observar la construcción del AFD, se pueden hacer las siguientes aclaraciones:

- **Estados inaccesibles (marcados con rojo):**

  Durante el recorrido del autómata se identificó que los siguientes estados **nunca pueden alcanzarse desde el estado inicial (q0,p0)**:

  - **(q0,p1)**
  - **(q1,p0)**
  - **(q1,p2)**

Esto se debe a la combinación de las transiciones de los AFD individuales:

| Estado inaccesible | Explicación |
|---|---|
| **(q0,p1)** | **Este estado implicaría que el autómata P ha leído un 1 (ya que p1 se alcanza solo desde p0 con un 1), pero que al mismo tiempo el autómata Q permanece en q0, lo cual no es posible. En Q, si el primer 1 aparece en una posición impar (como la primera letra de la cadena), se realiza una transición inmediata de q0 a q1. Por lo tanto, siempre que P llegue a p1, Q necesariamente abandona q0, imposibilitando la existencia de la combinación (q0,p1).** |
| **(q1,p0)** | **En este caso, se requeriría que Q haya detectado un 1 en una posición impar (condición para alcanzar q1), mientras que P no haya leído ningún 1 aún (por estar en p0). Esta combinación es contradictoria, ya que la única forma de que Q avance a q1 es mediante la lectura de un 1, lo cual automáticamente provoca en P la transición de p0 a p1. Así, no existe ninguna cadena que produzca simultáneamente estas dos condiciones, por lo que el estado (q1,p0) no es alcanzable.** |
| **(q1,p2)** | **El estado es inaccesible ya que, no hay ninguna secuencia de entrada que permita alcanzar simultáneamente estar en una posición par (lo que implica estar en q1) y haber leído exactamente dos 1s consecutivos (estado p2). En efecto, para llegar a p2 se requiere haber leído un segundo 1 en una posición par, lo que necesariamente hace que el autómata de las posiciones impares vuelva a q0. Por lo tanto, los estados q1 y p2 nunca coinciden en una misma configuración del autómata.** |

- **Estados aceptación:**

  El autómata producto tiene exactamente **dos estados de aceptación**:

  - **(q0,p3)**
  - **(q1,p3)**

  Esto se explica por los requisitos del lenguaje:

  - **p3** garantiza que la cadena **termina con la subcadena 111**.
  - **q0** o **q1** indican que **todas las posiciones impares contienen un 1**, que es lo requerido por el primer autómata.

  En cambio, los estados que incluyan **q2** no son de aceptación, ya que este estado representa haber leído al menos un **0** en una posición impar, **violando así una de las condiciones del lenguaje**.

4. Para cada AFD Brindar un ejemplo de una cadena que acepta y otra que rechaza. Explicar cómo determinar su autómata si es aceptado o rechazado paso a paso

- Lenguaje simple $L_{a1}$:

  - Cadena que acepta: "101110" → $q_0$ → $q_1$ → $q_2$ → $q_3$ → $q_4$ → $q_4$ → $q_4$ → "aceptado"

  Paso a paso: 101110 → $q_0$ – 1 → $q_1$ – 0 → $q_2$ – 1 → $q_3$ – 1 → $q_4$ – 1 → $q_4$ – 0 → $q_4$ → "aceptado"

  - Cadena que rechaza: "1010" → $q_0$ → $q_1$ → $q_2$ → $q_3$ → $q_1$ → "rechazado"

  Paso a paso: 1010 → $q_0$ – 1 → $q_1$ – 0 → $q_2$ – 1 → $q_3$ – 0 → $q_2$ → "rechazado" (no tiene dentro la cadena "1011" ni llega a $q_4$)

- Lenguaje simple $L_{a2}$:

  - Cadena que acepta: "1010" → $q_0$ → $q_0$ → $q_1$ → $q_1$ → $q_0$ → "aceptado"

  Paso a paso: 1010 → $q_0$ – 1 → $q_0$ – 0 → $q_1$ – 1 → $q_1$ – 0 → $q_0$ → "aceptado"

  - Cadena que rechaza: "1000" → $q_0$ → $q_0$ → $q_1$ → $q_0$ → $q_1$ → "rechazado"

  Paso a paso: 1000 → $q_0$ – 1 → $q_0$ – 0 → $q_1$ – 0 → $q_0$ – 0 → $q_1$ → "rechazado" (no tiene dentro de la cadena un número par de "0" ni llega a $q_0$)

- Lenguaje simple $L_{b1}$:

  - Cadena que acepta: "10101" → $q_0$ → $q_1$ → $q_0$ → $q_1$ → $q_0$ → $q_1$ → "aceptado"

  Paso a paso: 10101 → $q_0$ – 1 → $q_1$ – 0 → $q_0$ – 1 → $q_1$ – 0 → $q_0$ – 1 → $q_1$ → "aceptado"

  - Cadena que rechaza: "011" → $q_0$ → $q_2$ → $q_2$ → $q_2$ → "rechazado"

  Paso a paso: 011 → $q_0$ – 0 → $q_2$ – 1 → $q_2$ – 1 → $q_2$ → "rechazado" (no tiene "1" en toda posición impar ni llega a $q_1$)

- Lenguaje simple $L_{b2}$:

  - Cadena que acepta: "101111" → $q_0$ → $q_1$ → $q_0$ → $q_1$ → $q_2$ → $q_3$ → $q_3$ → "aceptado"

  Paso a paso: 101111 → $q_0$ – 1 → $q_1$ – 0 → $q_0$ – 1 → $q_1$ – 1 → $q_2$ – 1 → $q_3$ – 1 → $q_3$ → "aceptado"

  - Cadena que rechaza: "1110" → $q_0$ → $q_1$ → $q_2$ → $q_3$ → $q_0$ → "rechazado"

  Paso a paso: 1110 → $q_0$ – 1 → $q_1$ – 1 → $q_2$ – 1 → $q_3$ – 0 → $q_0$ → "rechazado" (no termina en "111" ni llega a $q_3$)

- Lenguaje complejo $L_a$:

- Cadena **aceptada:** "01001011"

**Estado inicial:** $(q_0, p_0)$

| **Entrada** | **Estado previo** | $\delta_1$ | $\delta_2$ | **Nuevo estado** |
|---|---|---|---|---|
| **0** | $(q_0, p_0)$ | $q_0 \rightarrow q_0$ | $p_0 \rightarrow p_1$ | $(q_0, p_1)$ |
| **1** | $(q_0, p_1)$ | $q_0 \rightarrow q_1$ | $p_1 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_1, p_1)$ | $q_1 \rightarrow q_2$ | $p_1 \rightarrow p_0$ | $(q_2, p_0)$ |
| **0** | $(q_2, p_0)$ | $q_2 \rightarrow q_0$ | $p_0 \rightarrow p_1$ | $(q_0, p_1)$ |
| **1** | $(q_0, p_1)$ | $q_0 \rightarrow q_1$ | $p_1 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_1, p_1)$ | $q_1 \rightarrow q_2$ | $p_1 \rightarrow p_0$ | $(q_2, p_0)$ |
| **1** | $(q_2, p_0)$ | $q_2 \rightarrow q_3$ | $p_0 \rightarrow p_0$ | $(q_3, p_0)$ |
| **1** | $(q_3, p_0)$ | $q_3 \rightarrow q_4$ | $p_0 \rightarrow p_0$ | $(q_4, p_0)$ |

**Estado final: $(q_4, p_0)$ → Aceptada**

- Cadena que **rechaza:** "0101011"

**Estado inicial:** $(q_0, p_0)$

| **Entrada** | **Estado previo** | $\delta_1$ | $\delta_2$ | **Nuevo estado** |
|---|---|---|---|---|
| **0** | $(q_0, p_0)$ | $q_0 \rightarrow q_0$ | $p_0 \rightarrow p_1$ | $(q_0, p_1)$ |
| **1** | $(q_0, p_1)$ | $q_0 \rightarrow q_1$ | $p_1 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_1, p_1)$ | $q_1 \rightarrow q_2$ | $p_1 \rightarrow p_0$ | $(q_2, p_0)$ |
| **1** | $(q_2, p_0)$ | $q_2 \rightarrow q_3$ | $p_0 \rightarrow p_0$ | $(q_3, p_0)$ |
| **0** | $(q_3, p_0)$ | $q_3 \rightarrow q_2$ | $p_0 \rightarrow p_1$ | $(q_2, p_1)$ |
| **1** | $(q_2, p_1)$ | $q_2 \rightarrow q_3$ | $p_1 \rightarrow p_1$ | $(q_3, p_1)$ |
| **1** | $(q_3, p_1)$ | $q_3 \rightarrow q_4$ | $p_1 \rightarrow p_1$ | $(q_4, p_1)$ |

**Estado Final: $(q_4, p_1)$ → No aceptado**

**Hay un error: cantidad impar de ceros**

- Lenguaje complejo $L_b$:

- Cadena **aceptada:** "1010111"

  **Estado inicial:** $(q_0, p_0)$

| **Entrada** | **Estado previo** | **$\delta_1$** | **$\delta_2$** | **Nuevo estado** |
|---|---|---|---|---|
| **1** | $(q_0, p_0)$ | $q_0 \rightarrow q_1$ | $p_0 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_1, p_1)$ | $q_1 \rightarrow q_0$ | $p_1 \rightarrow p_0$ | $(q_0, p_0)$ |
| **1** | $(q_0, p_0)$ | $q_0 \rightarrow q_1$ | $p_0 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_1, p_1)$ | $q_1 \rightarrow q_0$ | $p_1 \rightarrow p_0$ | $(q_0, p_0)$ |
| **1** | $(q_0, p_0)$ | $q_0 \rightarrow q_1$ | $p_0 \rightarrow p_1$ | $(q_1, p_1)$ |
| **1** | $(q_1, p_1)$ | $q_1 \rightarrow q_0$ | $p_1 \rightarrow p_2$ | $(q_0, p_2)$ |
| **1** | $(q_0, p_2)$ | $q_0 \rightarrow q_1$ | $p_2 \rightarrow p_3$ | $(q_1, p_3)$ |

  **Estado final: $(q_1, p_3)$ → Aceptada**

- Cadena **aceptada:** "1111"

  **Estado inicial:** $(q_0, p_0)$

| **Entrada** | **Estado previo** | **$\delta_1$** | **$\delta_2$** | **Nuevo estado** |
|---|---|---|---|---|
| **1** | $(q_0, p_0)$ | $q_0 \rightarrow q_1$ | $p_0 \rightarrow p_1$ | $(q_1, p_1)$ |
| **1** | $(q_1, p_1)$ | $q_1 \rightarrow q_0$ | $p_1 \rightarrow p_2$ | $(q_0, p_2)$ |
| **1** | $(q_0, p_2)$ | $q_0 \rightarrow q_1$ | $p_2 \rightarrow p_3$ | $(q_1, p_3)$ |
| **1** | $(q_1, p_3)$ | $q_1 \rightarrow q_0$ | $p_3 \rightarrow p_3$ | $(q_0, p_3)$ |

  **Estado final: $(q_0, p_3)$ → Aceptada**

- Cadena que **rechaza:** "10100111"

**Estado inicial:** $(q_0, p_0)$

| **Entrada** | **Estado previo** | $\delta_1$ | $\delta_2$ | **Nuevo estado** |
|---|---|---|---|---|
| **1** | $(q_0, p_0)$ | $q_0 \rightarrow q_1$ | $p_0 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_0, p_1)$ | $q_1 \rightarrow q_0$ | $p_1 \rightarrow p_0$ | $(q_0, p_0)$ |
| **1** | $(q_0, p_0)$ | $q_0 \rightarrow q_1$ | $p_0 \rightarrow p_1$ | $(q_1, p_1)$ |
| **0** | $(q_1, p_1)$ | $q_1 \rightarrow q_0$ | $p_1 \rightarrow p_0$ | $(q_0, p_0)$ |
| **0** | $(q_0, p_0)$ | $q_0 \rightarrow q_2$ | $p_0 \rightarrow p_0$ | $(q_2, p_0)$ |
| **1** | $(q_2, p_1)$ | $q_2 \rightarrow q_2$ | $p_0 \rightarrow p_1$ | $(q_2, p_1)$ |
| **1** | $(q_2, p_1)$ | $q_2 \rightarrow q_2$ | $p_1 \rightarrow p_2$ | $(q_2, p_2)$ |
| **1** | $(q_2, p_1)$ | $q_2 \rightarrow q_2$ | $p_2 \rightarrow p_3$ | $(q_2, p_3)$ |

**Estado Final: $(q_2, p_3)$ → No aceptado**

**Hay un error: no hay "1" en todas las posiciones impares ("0" en posición 5)**

# Referencias


Bertsimas, D., & Tsitsiklis, J. N. (1997). An Introduction to Linear Optimization. Athena Scientific.

Wolsey, L. A. (1998). Integer Programming. Wiley-Interscience.