.UBAfiuba
FACULTAD DE INGENIERÍA

TRABAJO PRÁCTICO 3

TEORÍA DE ALGORITMOS

(75.29 / 95.06 / TB024)

Cuatrimestre: 1º Cuatrimestre 2025

Fecha de entrega: 21 / 7 / 2025

Grupo: 1 (Los Desviados)

Curso: 3 - Echevarría

Integrantes:

Brian, Céspedes	108219
Mario, Janampa	108344
Juan, Juárez	110418
Cristian, Ledesma	111426

Índice

1. PROBLEMA 1

1.1. Instancia Particular

Tomando:

A = [1, 1, 1, 1, 9].
B = 9.

El algoritmo expuesto agrega los primeros 4 números, pues en cada iteración la suma es de 1, 2, 3 y 4 respectivamente.

$S_{algoritmoinicial}$ = [1, 1, 1, 1], cuya suma es de 4.
$S_{óptimo}$ = [9], que es el resultado de tomar únicamente el último elemento (suma 9).

$\frac{Suma\ Algoritmo}{Suma\ Óptimo} = \frac{4}{9} < \frac{1}{2}$.

Por lo tanto, se verifica que con dicha instancia de A y B, obtenemos un resultado que es menor que la mitad de la suma total de otro conjunto factible de A (en este caso, el conjunto de la solución óptima).

1.2. Diseño y Desarrollo del Algoritmo de Aproximación

Se diseña un algoritmo, el cual aprovecha el previamente otorgado en el enunciado. Además, se agrega una simple pero muy efectiva modificación: se ordena la lista de números disponibles (de mayor a menor) y se trabaja sobre dicha estructura.

Para poner de ejemplo, utilicemos la misma lista y número objetivo del ítem anterior:

A = [1, 1, 1, 1, 9].
B = 9.

Como ya vimos anteriormente, la solución del algoritmo greedy original toma los primeros 4 números de la lista, lo que nos da una suma total de 4.
Ahora, luego de ordenar la lista A de manera descendente, queda de la siguiente forma:

A = [9, 1, 1, 1, 1].

El algoritmo greedy simplemente tomaría el primer 9, y luego los números restantes se ignoran, debido a que exceden al valor de B (9).

$S_{algoritmomodificado}$ = [9], con una suma de 9 (en este caso es lo óptimo).

1.3. Supuestos

Se utilizan listas en vez de conjuntos (para admitir duplicados).
La lista de números disponibles no contiene negativos.
La lista de números disponibles posee números enteros.
La lista de números disponibles no está vacía.
El número objetivo es entero y positivo.

1.4. Diseño

a) Enunciando cómo se cumple la garantía solicitada:

En el paper “A fast approximation algorithm for the subset-sum problem” (ver referencias bibliográficas) se aborda sobre el problema conocido como “Subset-Sum”. El problema de optimización de Subset-Sum, se enuncia: dado un conjunto de números enteros positivos $A = \{x_1, x_2, \ldots, x_n\}$ y un número entero positivo B. Encontrar un conjunto $S \subseteq A$ tal que

$$\sum_{x \in S} x \leq B$$

Esto es exactamente lo que nos pide el enunciado del presente ejercicio. En el paper anteriormente mencionado, se indica que el algoritmo propuesto que ordena los elementos descendentemente garantiza al menos la mitad del valor óptimo.

En el libro "Knapsack Problems" de Martello & Toth (ver referencias bibliográficas), se menciona que si se considera sólo el mayor número como posible solución alternativa, se garantiza que la peor solución posible será al menos el 50% del óptimo (página 127). Sin embargo, para obtener mejores resultados, en promedio, conviene ordenar los elementos de mayor a menor antes de aplicar el algoritmo (esto permite evitar calcular explícitamente cuál es el número más grande porque ya estará primero en la lista).

b) Pseudocódigo e Implementación:

Pseudocódigo

```
mejor_subconjunto_aproximado(numeros_disponibles, numero_objetivo):

    ORDENAR numeros_disponibles DESCENDENTEMENTE

    elementos_elegidos = lista_vacía()
    suma = 0

    PARA CADA numero EN numeros_disponibles:
        SI suma + numero <= numeros_disponibles ENTONCES:
            AGREGAR numero A elementos_elegidos
            suma += numero

    RETORNAR elementos_elegidos, suma
```

Implementación

```python
def mejor_subconjunto_aproximado(numeros_disponibles: list, numero_objetivo: int) -> tuple:

    numeros_disponibles.sort(reverse=True)

    elementos_elegidos: list = list()
    suma: int = 0

    for numero in numeros_disponibles:
        if suma + numero <= numero_objetivo:
                elementos_elegidos.append(numero)
                suma += numero

    return elementos_elegidos, suma
```

c) Estructuras de Datos Utilizadas:

Las estructuras de datos utilizadas son las mismas que en el algoritmo inicial. La lista de números disponibles (anteriormente llamada A) y por último, los elementos elegidos (anteriormente llamado S).

1.5. Seguimiento

Para hacer un seguimiento, se plantea la siguiente lista de números y el siguiente número objetivo a alcanzar:

Lista de Números = [7, 10, 5, 21, 2].
Número Objetivo = 25.

Para comparar luego, es claro notar que el óptimo es alcanzable con S = [7, 10, 5, 2] (se adelanta que nuestro algoritmo no hallará dicho óptimo, pero encontrará una aproximación muy buena).

Inicialmente, se ordena la lista de números de manera descendente, quedando de la siguiente manera:

Lista de Números = [21, 10, 7, 5, 2].

Ahora, se realiza el algoritmo greedy propuesto. Inicialmente la suma es de 0, por lo que el número 21 se agrega a la lista, ya que el mismo es menor que el límite (25).

$S_{parcial1}$ = [21].
$T_{parcial1}$ = 21.

Luego, se intenta agregar el 10, pero si lo hiciéramos, 21 + 10 = 31 > 25, por lo que se ignora dicho número.

Al igual que en el caso anterior, se intenta agregar el 7, pero 21 + 7 = 28 > 25, se descarta dicho número.

Pasa lo mismo al intentar agregar el 5 ya que 21 + 5 = 26 > 25, no agregamos el 5.

Finalmente, al llegar al 2, verificamos que 21 + 2 = 23, por lo que sí podemos agregar dicho número, quedando al final:

S = [21, 2].
T = 23.

Sin embargo, la solución óptima ya habíamos anticipado que era:

$S_{óptimo}$ = [10, 7, 5, 2].
$T_{óptimo}$ = 24 (que se acerca más a 25 aún).

$\frac{23}{24} \approx 0,96 \geq 0,5$

Se verifica que la solución que encontró el algoritmo greedy es mayor o igual a la mitad del óptimo verdadero.

1.6. Complejidad

Analizaremos la complejidad de la única función previamente vista:

La función **mejor_subconjunto_aproximado()** realiza un ordenamiento sobre la lista de números disponibles, y posteriormente, con el enfoque greedy se recorre la lista en su totalidad, en búsqueda de la mejor suma posible.

- **Complejidad resultante (temporal):** $O(n \log n) + O(n) = O(n \log n)$**.**

La complejidad espacial está principalmente dada por la lista de los elementos obtenidos (o sea, el listado de números que debemos sumar). Esta lista, en el peor caso, tendrá los mismos elementos que la lista de números original.

- **Complejidad resultante (espacial):** $O(n)$.

1.7. Sets de Datos

En el archivo **test_aleatorio.txt** se encuentran las pruebas realizadas sobre los sets de datos generados aleatoriamente con el programa **ejercicio1_test_aleatorio.py**.

Para medir los tiempos de ejecución, se optó por utilizar otro programa **ejercicio1_test_tiempo_de_ejecucion.py**. Los datos obtenidos se pueden visualizar en el archivo **test_tiempos_de_ejecucion_entrega.txt**. Sobre este archivo se realizó el gráfico que se verá en la siguiente sección. Si se corre nuevamente el script, se escribirá un archivo **test_tiempos_de_ejecucion.txt**, para comparar los resultados obtenidos con los de la entrega.

Como tercera opción, se encuentra el programa **ejercicio1_test_manual.py** el cual permite al usuario introducir un número objetivo y una lista de números a demanda y luego probar el algoritmo.

Finalmente, para probar qué tan bueno es el algoritmo en comparación con la solución exacta, se utilizó el programa **ejercicio1_test_garantia.py**. En el mismo, se permite seleccionar la cantidad de iteraciones que realizará el algoritmo y la longitud de la lista (no se recomienda una longitud de lista excesiva, debido a que la complejidad del algoritmo que halla la mejor suma exacta es exponencial, y podría demorar demasiado tiempo en dar una solución).

Al final de dicho programa, se presenta un caso borde en el cual tiende a alcanzarse el límite del 50% de la optimalidad.

1.8. Tiempos de Ejecución

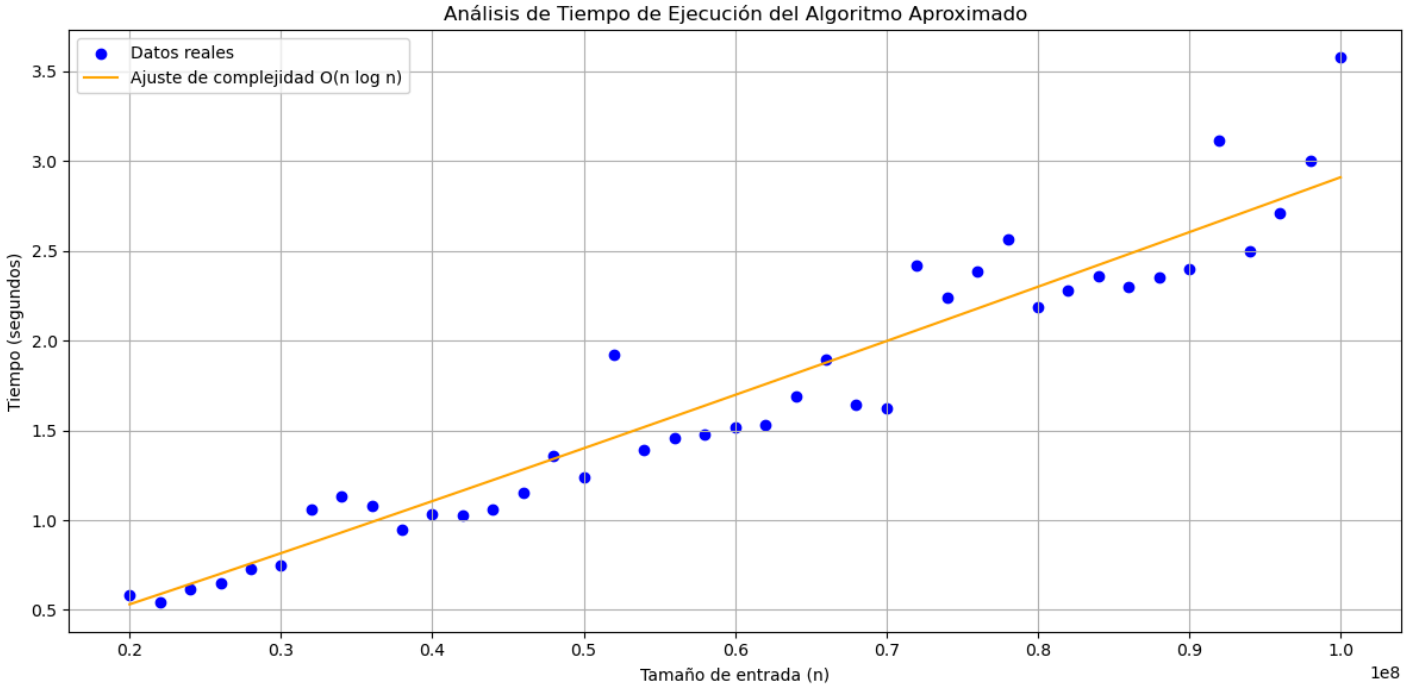


Gráfico que relaciona la cantidad de números n de con el tiempo de ejecución *

** Las mediciones fueron realizadas en una computadora con las siguientes características:*

- **Procesador:** [*Intel i5 1235U, 1.3GHz*].
- **Memoria RAM:** [16 GB].
- **Sistema Operativo:** [Windows 11].
- **Intérprete de Python:** [Python 3.12.3].

1.9. Informe de Resultados

Podemos concluir que el algoritmo presentado se aproxima correctamente a la solución óptima. Lo anterior, además, se realiza en un tiempo considerable, ya que el algoritmo presenta una complejidad de O(n log n).

Si bien el algoritmo no da siempre la solución óptima, se probó que la solución factible obtenida es al menos la mitad de la misma. Se citan las referencias bibliográficas correspondientes, y también se agrega una prueba empírica para reforzar lo enunciado.

2. PROBLEMA 2

2.1. Diseño y Desarrollo del Algoritmo Randomizado

El problema busca diseñar una estrategia donde el vendedor reciba ofertas en un orden aleatorio y tenga una probabilidad de al menos ¼ de aceptar la mejor de las n ofertas. Este comparte su naturaleza con el conocido Problema del Secretario, o también llamado Problema de la Mejor Elección.

Sus similitudes se basan en el hecho de buscar al mejor candidato (dígase, ofertas o empleados) de una secuencia con un orden aleatorio, donde la decisión de seleccionar o avanzar son irreversibles y teniendo conocimiento únicamente de información pasada.

Este problema es abordado en el trabajo de Ferguson (1989), donde se describe una estrategia de observación y posteriormente selección, logrando una probabilidad óptima de éxito de 1/e (aproximadamente 36.8%). Consiste de:

Estrategia del Problema del Secretario:

1. Rechazar las primeras r observaciones, pero recordando el mayor valor avistado.
2. A partir de la observación r + 1, aceptar la primera observación mayor al mejor valor de la etapa 1.

En el planteo de este ejercicio, se desea elegir la mejor oferta de una secuencia de n compradores. El objetivo es seleccionar la mayor oferta con una probabilidad mayor a ¼.

Estrategia del ejercicio de algoritmos randomizados:

1. Rechazar las primeras m = [n/4] ofertas, recordando el mayor valor avistado (observación).
2. A partir de la oferta [m + 1], aceptar la primera oferta mayor al mejor valor de la etapa 1. En caso de no encontrar una mejor, se acepta la última.

La estrategia óptima sugerida por Hill utiliza r = [n/e] para maximizar la probabilidad de éxito, sin embargo, para adaptarnos a los requerimientos del enunciado, usando r = [n/4] y bajo las mismas condiciones que el paper (aleatoriedad, decisiones irrevocables, memoria de valores anteriores), la probabilidad de elegir la mayor oferta es mayor o igual a ¼.

En conclusión, el algoritmo descrito en el paper de Hill, se adapta al problema del ejercicio con una modificación en el valor de r, cumpliéndo la condición probabilística solicitada en el enunciado.

2.2. Supuestos

- Las ofertas llegan en un orden aleatorio.
- Las ofertas llegan en forma de una lista de enteros con valores mayores a 0.
- Se conoce el valor de n previamente.
- Las decisiones sobre las ofertas son irreversibles.

- Se debe elegir una oferta estrictamente.

2.3. Diseño

Pseudocódigo

```
def elegir_mejor_oferta(array ofertas):
   n -> tamaño de ofertas
   m -> n / 4 (redondeado hacia arriba)
   mejor_oferta -> inicializada en 0

   desde i = 0 hasta m:
      si ofertas[i] > mejor_oferta:
         mejor_oferta = ofertas[i]

   desde j = m + 1 hasta n:
      si ofertas[j] > mejor_oferta:
         retornar ofertas[j]

   retornar ofertas[-1]
```

Estructuras utilizadas:

- Lista de enteros (ofertas) para representar las ofertas.
- Variable entera (mejor_oferta) para almacenar la mejor oferta observada.

Implementación

```python
def elegir_mejor_oferta(ofertas):
   n = len(ofertas)
   m = floor(n / 4)
   mejor_oferta = 0

   for i in range(m):
      if ofertas[i] > mejor_oferta:
         mejor_oferta = ofertas[i]

   for j in range(m, n):
      if ofertas[j] > mejor_oferta:
         return ofertas[j]

   return ofertas[-1]
```

2.4. Seguimiento

1. Valores Iniciales:
   ofertas = [20, 10, 25, 30, 50, 10, 30, 80]

n = len(ofertas) = 8
m = n/4 = 2
mejor_oferta = 0

2. Observación:
   desde i = 0 hasta m: (0, 1):
   - i = 0:
     ofertas[0] = 20 -> mayor a mejor_oferta
     mejor_oferta = 20
   - i = 1:
     ofertas[1] = 10 -> menor a mejor_oferta
     mejor_oferta = 20

3. Selección:
   desde j = 2 hasta n: (2, …, 7)
   - j = 2:
     ofertas[2] = 25 -> mayor a mejor_oferta
     retorna ofertas[2] = 25

Resultado: La función acepta la oferta 25 y termina. No analiza el resto (30, 50, 10, 30, 80), aunque 80 sea la mejor de todas.

2.5. Complejidad

El algoritmo requiere recorrer las ofertas una sola vez, lográndolo en dos etapas secuenciales:

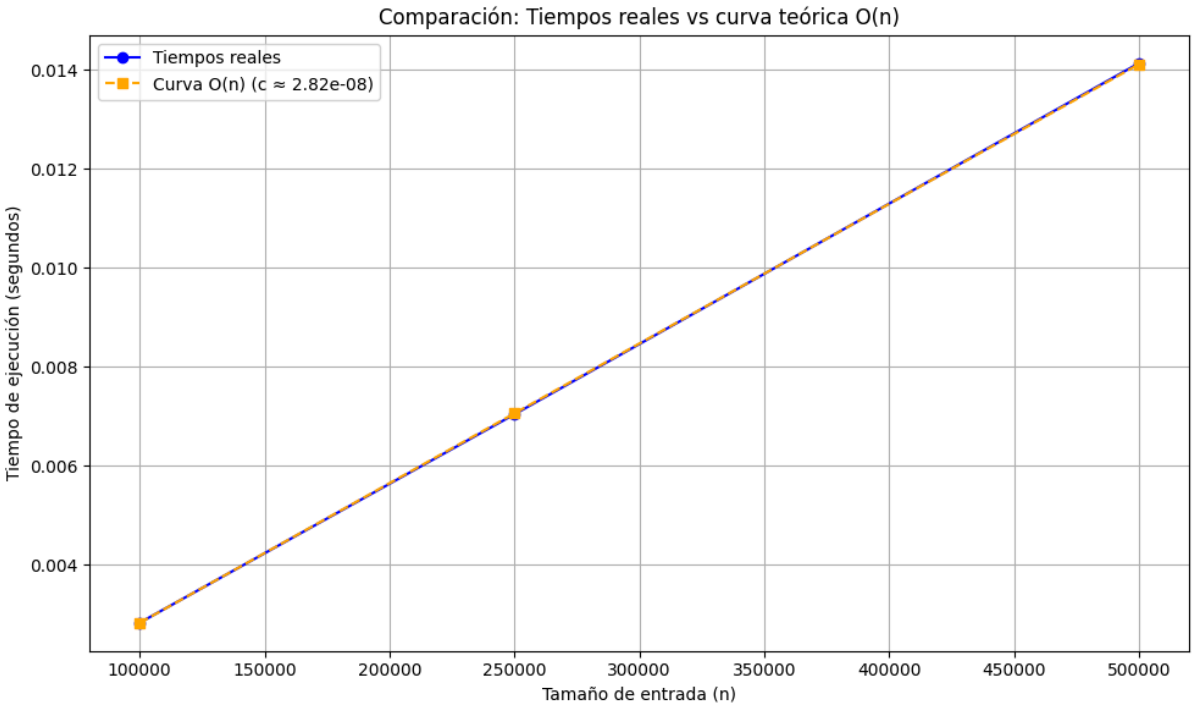
- Encontrar el máximo entre las primeras m ofertas -> O(n).
- Comparar desde m hasta n -> O(n).

Por lo cual, la complejidad total resulta lineal, siendo O(n).

2.6. Sets de Datos

Se utiliza la función generar_ofertas para la generación de ofertas aleatorias, la cual recibe por parámetro la cantidad de ofertas que se desean generar y devuelve un array de números enteros, donde cada una representa una oferta. Se hace uso de la librería random junto con la semilla “6” para garantizar su reproducibilidad. Se generan con tamaños de 100.000, 250.000 y 500.000 para poder medir los tiempos de ejecución.

2.7. Tiempos de Ejecución



Tamaño	Tiempo	Factor de aumento (tamaño/tiempo)
100.000	0.00282631	1
250.000	0.00703866	2.5
500.000	0.01413631	5

2.8. Informe de Resultados

Se realizaron 1000 pruebas independientes, cada una con una lista de 1000 ofertas generadas aleatoriamente, usando una semilla distinta.

En cada prueba, se usó la función elegir_mejor_oferta(), y se comparó el resultado con la mejor oferta real de la lista.

Los resultados arrojaron que la mejor oferta se eligió 369 veces, y considerando que se realizaron 1000 pruebas, se calcula un porcentaje de aciertos del 36.90%.

El valor obtenido se encuentra por encima del 25% requerido. Esto respalda de manera empírica la estrategia basada en el Problema del Secretario, cumpliendo con la garantía solicitada de tener una probabilidad ≥ ¼ de seleccionar la mejor oferta.

Podemos observar que los resultados concuerdan con el análisis teórico basado en el paper de Hill acerca del problema del secretario, donde se alcanzaba probabilidades de éxito de al menos 1/e en el caso óptimo. Por lo cual, los resultados experimentales son consistentes con la teoría.

3. PROBLEMA 3

Dado nuestro problema de interés, el Alfabeto finito de entrada será {0, 1}.
Σ = {0, 1}.

3.1. Descripción Formal del Autómata (Lenguajes Simples)

1) Lenguaje simple A1 (comienza con “1”):

- Conjunto de estados:
  - Q1 = { p0, p1, p2 }.

- Función de transición en forma de tabla (δ1):

	0	1
p0	p2	p1
p1	p1	p1
p2	p2	p2

- Estado Inicial: { p0 }.
- Estados de aceptación: F1 = { p1 }.
- **A1 = (Q1, Σ , δ1, p0, F1).**

2) Lenguaje simple A2 (tiene cuanto mucho un “0”):

- Conjunto de estados:
  - Q2 = { r0, r1, r2 }.

- Función de transición en forma de tabla (δ2):

	0	1
r0	r1	r0
r1	r2	r1
r2	r2	r2

- Estado Inicial: { r0 }.
- Estados de aceptación: F2 = { r0, r1 }.
- **A2 = (Q2, Σ , δ2, r0, F2).**

3) Lenguaje simple B1 (tiene un número par de “1”):

- Conjunto de estados:
  - Q3 = { p0, p1 }.

- Función de transición en forma de tabla (δ3):

	0	1
p0	p0	p1
p1	p1	p0

- Estado Inicial: { p0 }.
- Estados de aceptación: F3 = { p0 }.
- **B1 = (Q3, Σ , δ3, p0, F3).**

4) Lenguaje simple B2 (tiene uno o dos “0”):

- Conjunto de estados:
  - Q4 = { r0, r1, r2, r3 }.

- Función de transición en forma de tabla (δ4):

	0	1
r0	r1	r0
r1	r2	r1
r2	r3	r2
r3	r3	r3

- Estado Inicial: { r0 }.
- Estados de aceptación: F4 = { r1, r2 }.
- **B2 = (Q4, Σ , δ4, r0, F4).**

3.2. Descripción Formal de los Autómatas (Lenguajes A y B)

Para hallar los autómatas A y B, vamos a hacer el producto cartesiano entre sus lenguajes simples para poder obtener todos los recorridos que hace el autómata.

- Lenguaje A: QA = Q1 x Q2, { (p, r) / p ∈ Q1, r ∈ Q2 }.
  - QA = { (p0, r0), (p0, r1), (p0, r2), (p1, r0), (p1, r1), (p1, r2), (p2, r0), (p2, r1), (p2, r2) }.
- Lenguaje B: QB = Q3 x Q4, { (p, r) / p ∈ Q3, r ∈ Q4 }.

- QB = { (p0, r0), (p0, r1), (p0, r2), (p0, r3), (p1, r0), (p1, r1), (p1, r2), (p1, r3) }.

- Función de transición en forma de tabla (δA):

	0	1
(p0, r0)	(p2, r1)	(p1, r0)
(p0, r1)	(p2, r2)	(p1, r1)
(p0, r2)	(p2, r2)	(p1, r2)
(p1, r0)	(p1, r1)	(p1, r0)
(p1, r1)	(p1, r2)	(p1, r1)
(p1, r2)	(p1, r2)	(p1, r2)
(p2, r0)	(p2, r1)	(p2, r0)
(p2, r1)	(p2, r2)	(p2, r1)
(p2, r2)	(p2, r2)	(p2, r2)

- Haciendo cambio de variable:

(p0, r0) -> q0.
(p0, r1) -> q1.
(p0, r2) -> q2.
(p1, r0) -> q3.
(p1, r1) -> q4.
(p1, r2) -> q5.
(p2, r0) -> q6.
(p2, r1) -> q7.
(p2, r2) -> q8.

- Tabla de transiciones con cambio de variable:

	0	1
q0	q7	q3
q1	q8	q4
q2	q8	q5
q3	q4	q3
q4	q5	q4
q5	q5	q5
q6	q7	q6
q7	q8	q7
q8	q8	q8

- Teniendo así como resultado:
  - Conjunto de Estados: QA = { q0, q1, q2, q3, q4, q5, q6, q7, q8 }.
  - Estado inicial = { q0 }.
  - Estados de aceptación: FA = { q3, q4 }.
  - **A = (QA, Σ , δA, q0, FA).**

- Función de transición en forma de tabla (δB):

	0	1
(p0, r0)	(p0, r1)	(p1, r0)
(p0, 1r)	(p0, r2)	(p1, r1)
(p0, r2)	(p0, r3)	(p1, r2)
(p0, r3)	(p0, r3)	(p1, r3)
(p1, r0)	(p1, r1)	(p0, r0)
(p1, r1)	(p1, r2)	(p0, r1)
(p1, r2)	(p1, r3)	(p0, r2)
(p1, r3)	(p1, r3)	(p0, r3)

- Haciendo cambio de variable:

(p0, r0) -> q0.
(p0, r1) -> q1.
(p0, r2) -> q2.
(p0, r3) -> q3.
(p1, r0) -> 14.
(p1, r1) -> q5.
(p1, r2) -> q6.
(p1, r3) -> q7.

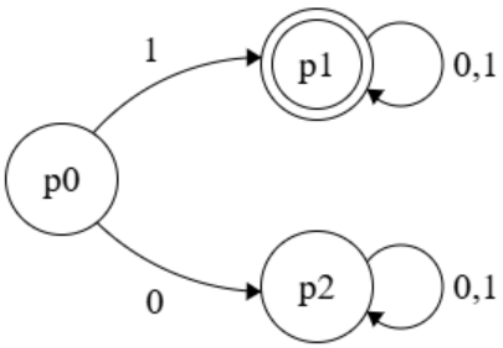
- Tabla de transiciones con cambio de variable:

	0	1
q0	q1	q4
q1	q2	q5
q2	q3	q6
q3	q3	q7
q4	q5	q0
q5	q6	q1
q6	q7	q2
q7	q7	q3

- Teniendo así como resultado:
  - Conjunto de Estados: Q = { q0, q1, q2, q3, q4, q5, q6, q7 }.
  - Estado inicial = { q0 }.
  - Estados de aceptación: F = { q1, q2 }.
  - **B = (QB, Σ , δB, q0, FB).**

3.3. Representación Gráfica y Ejemplos de Cadenas

Lenguaje Simple A1:



Ejemplo 1: “101”.

Estado inicial: p0.

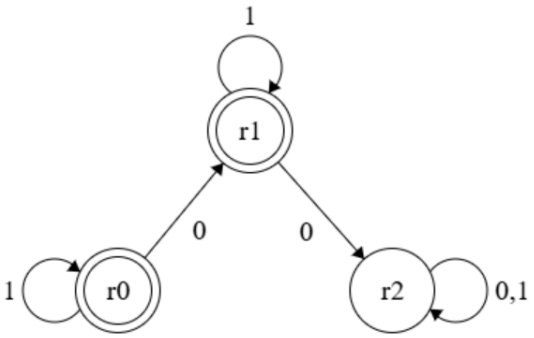
1) Se lee “1”: q0 -> p1.
2) Se lee “0”: p1 -> p1.
3) Se lee “1”: p1 -> p1.
4) Terminó en un estado de aceptación, la cadena es aceptada.

Ejemplo 2: “010”.

Estado inicial: p0.

1) Se lee “0”: p0 -> p2.
2) Se lee “1”: p2 -> p2.
3) Se lee “0”: p2 -> p2.
4) No terminó en un estado de aceptación, la cadena es rechazada.

Lenguaje Simple A2:



Ejemplo 1: “1011”.

Estado inicial: r0.

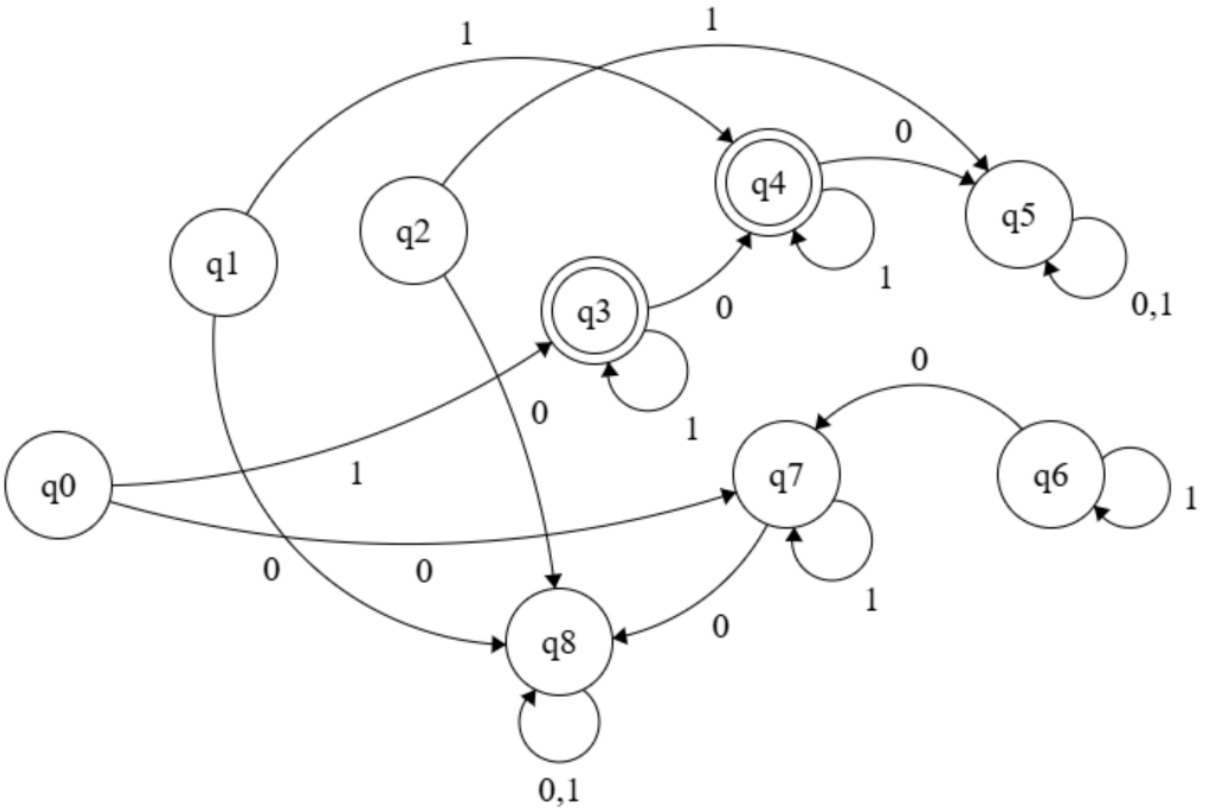
1) Se lee “1”: r0 -> r0.
2) Se lee “0”: r0 -> r1.
3) Se lee “1”: r1 -> r1.
4) Se lee “1”: r1 -> r1.
5) Terminó en estado de aceptación, la cadena es aceptada.

Ejemplo 2: “0101”.

Estado inicial: r0.

1) Se lee “0”: r0 -> r1.
2) Se lee “1”: r1 -> r1.
3) Se lee “0”: r1 -> r2.
4) Se lee “1”: r2 -> r2.
5) No terminó en estado de aceptación, la cadena es rechazada.

Lenguaje A:



Ejemplo 1: “11101”.

Estado inicial: q0.

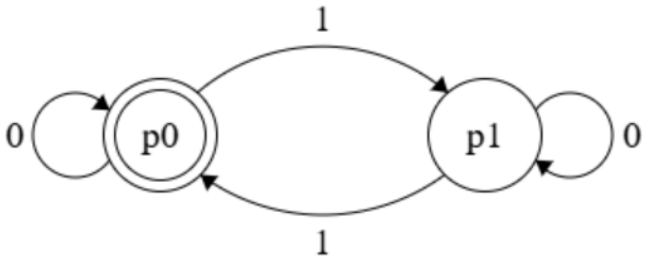
1) Se lee “1”: q0 -> q3.
2) Se lee “1”: q3 -> q3.
3) Se lee “1”: q3 -> q3.
4) Se lee “0”: q3 -> q4.
5) Se lee “1”: q4 -> q4.
6) Terminó en un estado de aceptación, la cadena es aceptada.

Ejemplo 2: “11010”.

Estado inicial: q0.

1) Se lee “1”: q0 -> q3.
2) Se lee “1”: q3 -> q3.
3) Se lee “0”: q3 -> q4.
4) Se lee “1”: q4 -> q4.
5) Se lee “0”: q4 -> q5.
6) No terminó en un estado de aceptación, la cadena es rechazada.

Lenguaje Simple B1



Ejemplo 1: “0110”.

Estado inicial: p0.

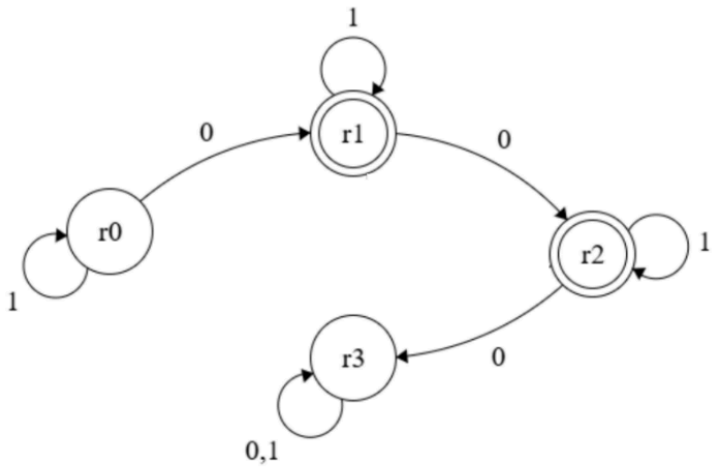
1) Se lee “0”: p0 -> p0.
2) Se lee “1”: p0 -> p1.
3) Se lee “1”: p1 -> p0.
4) Se lee “0”: p0 -> p0.
5) Terminó en un estado de aceptación, la cadena es válida.

Ejemplo 2: “1101”.

Estado inicial: p0.

1) Se lee “1”: p0 -> p1.
2) Se lee “1”: p1 -> p0.
3) Se lee “0”: p0 -> p0.
4) Se lee “1”: p0 -> p1.
5) No terminó en un estado de aceptación, la cadena es rechazada.

Lenguaje Simple B2:



Ejemplo 1: “1010”.

Estado inicial: r0.

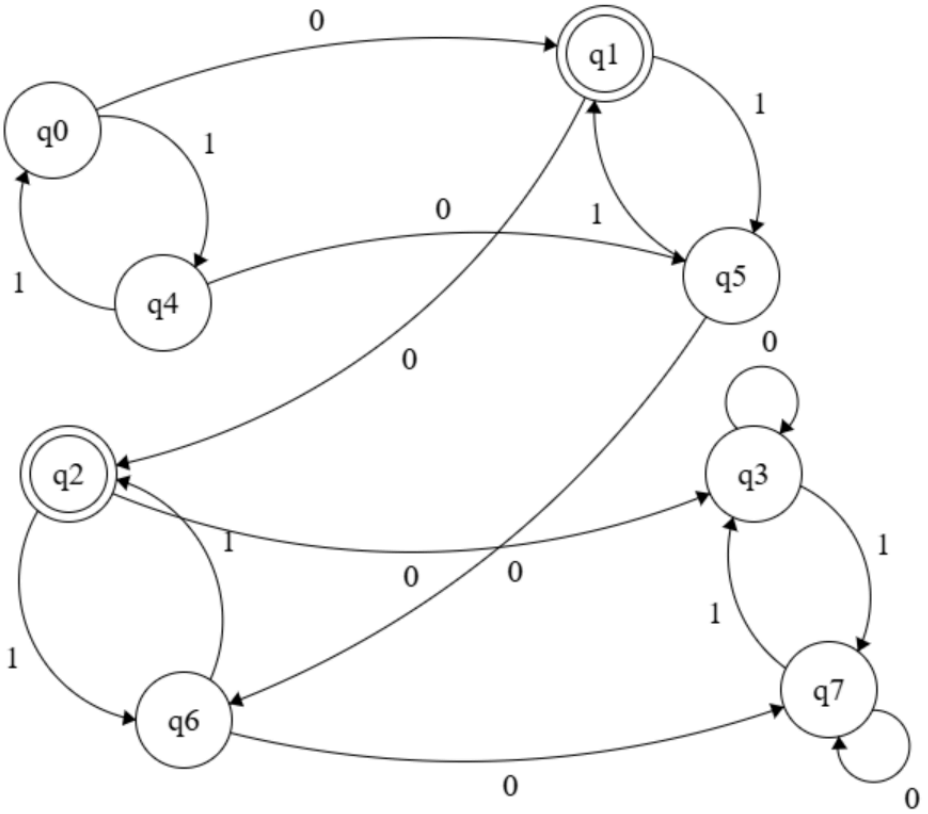
1) Se lee “1”: r0 -> r0.
2) Se lee “0”: r0 -> r1.
3) Se lee “1”: r1 -> r1.
4) Se lee “0”: r1 -> r2.
5) Terminó en un estado de aceptación, la cadena es válida.

Ejemplo 2: “0100”.

Estado inicial: r0.

1) Se lee “0”: r0 -> r1.
2) Se lee “1”: r1 -> r1.
3) Se lee “0”: r1 -> r2.
4) Se lee “0”: r2 -> r3.
5) No terminó en un estado de aceptación, la cadena es rechazada.

Lenguaje B:



Ejemplo 1: “011101”.

Estado inicial: q0.

1) Se lee “0”: q0 -> q1.
2) Se lee ‘1”: q1 -> q5.
3) Se lee “1”: q5 -> q1.
4) Se lee “1”: q1 -> q5.
5) Se lee “0”: q5 -> q6.
6) Se lee “1”: q6 -> q2.
7) Terminó en un estado de aceptación, la cadena es válida.

Ejemplo 2: “110100”.

Estado inicial: q0.

1) Se lee “1”: q0 -> q4.
2) Se lee “1”: q4 -> q0.
3) Se lee “0”: q0 -> q1.
4) Se lee “1”: q1 -> q5.
5) Se lee “0”: q5 -> q6.
6) Se lee “0”: q6 -> q7.
7) No terminó en un estado de aceptación, la cadena es rechazada.

REFERENCIAS

- Python Software Foundation. (s.f.). *Time complexity*. Python Wiki. https://wiki.python.org/moin/TimeComplexity
- Przydatek, B. (2002). *A fast approximation algorithm for the subset-sum problem*. Carnegie Mellon University. https://crypto.ethz.ch/publications/files/Przyda02.pdf
- Martello, S., & Toth, P. (1990). *Knapsack problems: Algorithms and computer implementations*. John Wiley & Sons.
- Ferguson, T. S. (1989). *Who Solved the Secretary Problem? Statistical Science.*

  https://www2.math.upenn.edu/~ted/210F10/References/Secretary.pdf