Índice

1. Problema 4

Para el Problema 4 se solicita desarrollar un algoritmo de Programación Dinámica que encuentre la secuencia de elementos contiguos que sume el máximo valor posible. El enfoque adoptado resuelve el problema mediante una iteración lineal sobre el array manteniendo sumas parciales óptimas.

1.1. Análisis

1.1.1. Supuestos, condiciones y limitaciones

- **Supuestos:** El algoritmo asume que la entrada es un array (o lista) de elementos numéricos que soportan operaciones de suma y comparación.
- **Condiciones:** Funciona con arrays no ordenados que pueden contener números positivos, negativos y/o ceros.
- **Limitaciones:** La implementación está diseñada para devolver una única subsecuencia. Si existen múltiples subsecuencias con la misma suma máxima, el algoritmo devolverá la primera que sea completamente evaluada.

1.1.2. Ecuación de Recurrencia

La ecuación de recurrencia define la suma máxima de un subarray que termina en el índice i como:

$$dp[i] = \text{máx}(A[i], dp[i-1] + A[i]) \tag{1}$$

donde $dp[i]$ representa la suma máxima del subarray que termina en i. La suma máxima global es el máximo valor en dp.

1.1.3. Justificación del cumplimiento de los requisitos de Subestructura Óptima y Subproblemas Superpuestos

- **Subestructura Óptima:** La solución óptima para un subarray que termina en i se construye a partir de la solución óptima del subarray que termina en $i-1$. Si extendemos el subarray anterior añadiendo $A[i]$, o empezamos uno nuevo en $A[i]$, garantizamos que la elección local lleva a la solución global óptima, ya que cada decisión considera la mejor suma previa.
- **Subproblemas Superpuestos:** Los subproblemas (sumas máximas terminando en cada índice) se reutilizan en la iteración. La acumulación de sumas parciales evita recalcular sumas desde cero, resolviendo subproblemas en orden lineal.

1.1.4. Indicación de cómo usa Memoization

La implementación utiliza la variable `current_sum` que actúa como memorización de la suma parcial optima anterior. Esto evita recalcular sumas previas y resuelve el problema en una sola pasada.

1.2. Diseño

1.2.1. Pseudocódigo

```
FUNCION encontrar_maxima_subsecuenci(Array A):
    n = longitud(A)
    SI n == 0:
        DEVOLVER (0, 0, 0)

    max_suma = A[0]
    max_inicio = 0
    max_fin = 0
    suma_actual = A[0]
    inicio_actual = 0

    PARA i DESDE 1 HASTA n-1:
        SI A[i] > suma_actual + A[i]:
            suma_actual = A[i]
            inicio_actual = i
        SINO:
            suma_actual = suma_actual + A[i]

        SI suma_actual > max_suma:
            max_suma = suma_actual
            max_inicio = inicio_actual
            max_fin = i

    DEVOLVER (max_suma, max_inicio, max_fin)
```

Listing 1: Pseudocódigo del algoritmo de Programación Dinámica

A continuación, se muestra el código realizado en Python:

```python
def subsequence_max(arr):
    if not arr:  # Manejar array vac o
        return 0, 0, 0

    # Inicializaci n con el primer elemento
    max_sum = arr[0]
    max_start = 0
    max_end = 0
    current_sum = arr[0]
    current_start = 0

    # Iterar desde el segundo elemento
    for i in range(1, len(arr)):
        # Decidir si empezar nuevo subarray o extender el actual
        if arr[i] > current_sum + arr[i]:
            current_sum = arr[i]
            current_start = i
        else:
            current_sum += arr[i]

        # Actualizar la soluci n global si encontramos una suma mayor
        if current_sum > max_sum:
            max_sum = current_sum
            max_start = current_start
            max_end = i

    return max_sum, max_start, max_end
```

Listing 2: Código de la función para hallar la secuencia de uno o más elementos contiguos que sumen el máximo valor posible mediante Programación Dinámica

1.2.2. Estructuras de datos utilizadas

- **Array/Lista:** Se utiliza para almacenar los datos de entrada. Su elección es natural, ya que permite un acceso eficiente en tiempo constante ($O(1)$) a los elementos por su índice.
- **Tupla:** Se utilizan para almacenar `max_suma`, `max_start` y `max_end` al momento de haber encontrado la mejor solución.

1.3. Seguimiento

Se utiliza el siguiente set de datos: `A = [2, -4, 3, 1]`. **Ejecución paso a paso:**

1. **Inicialización:**
   - `max_suma = A[0] = 2`
   - `max_inicio = 0`, `max_fin = 0`
   - `suma_actual = A[0] = 2`
   - `inicio_actual = 0`
2. **i = 1 (A[1] = -4):**
   - ¿`A[1] >suma_actual + A[1]`? → ¿`-4 >2 + (-4) = -2`? → No
   - `suma_actual = suma_actual + A[1] = 2 + (-4) = -2`
   - ¿`suma_actual >max_suma`? → ¿`-2 >2`? → No
   - Estado: `max_suma = 2, suma_actual = -2, inicio_actual = 0`
3. **i = 2 (A[2] = 3):**
   - ¿`A[2] >suma_actual + A[2]`? → ¿`3 >-2 + 3 = 1`? → Sí
   - `suma_actual = A[2] = 3`, `inicio_actual = 2`
   - ¿`suma_actual >max_suma`? → ¿`3 >2`? → Sí
   - Actualizar: `max_suma = 3, max_inicio = 2, max_fin = 2`
4. **i = 3 (A[3] = 1):**
   - ¿`A[3] >suma_actual + A[3]`? → ¿`1 >3 + 1 = 4`? → No
   - `suma_actual = suma_actual + A[3] = 3 + 1 = 4`
   - ¿`suma_actual >max_suma`? → ¿`4 >3`? → Sí
   - Actualizar: `max_suma = 4, max_inicio = 2, max_fin = 3`

Resultado Final:

- La suma máxima es **4**
- Corresponde a la subsecuencia `A[2:4] = [3, 1]` (índices 2 y 3)
- La subsecuencia va desde el índice `max_inicio = 2` hasta `max_fin = 3`

1.4. Complejidad

El análisis de la complejidad temporal se realiza a partir del pseudocódigo. Hay un único bucle que recorre el array de tamaño n, con operaciones internas de tiempo constante. Por lo tanto, la complejidad del algoritmo es $\mathbf{O(n)}$.

1.5. Sets de Datos

Se proponen los siguientes sets de datos para verificar la correctitud del algoritmo:

- **Caso General:** `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`
  Resultado: Suma = 6, Subseq = `[4, -1, 2, 1]`
- **Todos Positivos:** `[1, 2, 3, 4]`
  Resultado: Suma = 10, Subseq = `[1, 2, 3, 4]`
- **Todos Negativos:** `[-5, -1, -3]`
  Resultado: Suma = -1, Subseq = `[-1]`
- **Caso "Sándwich":** `[-10, 4, 2, -5, 8, -20]`
  Resultado: Suma = 9, Subseq = `[4, 2, -5, 8]`

1.6. Tiempos de ejecución

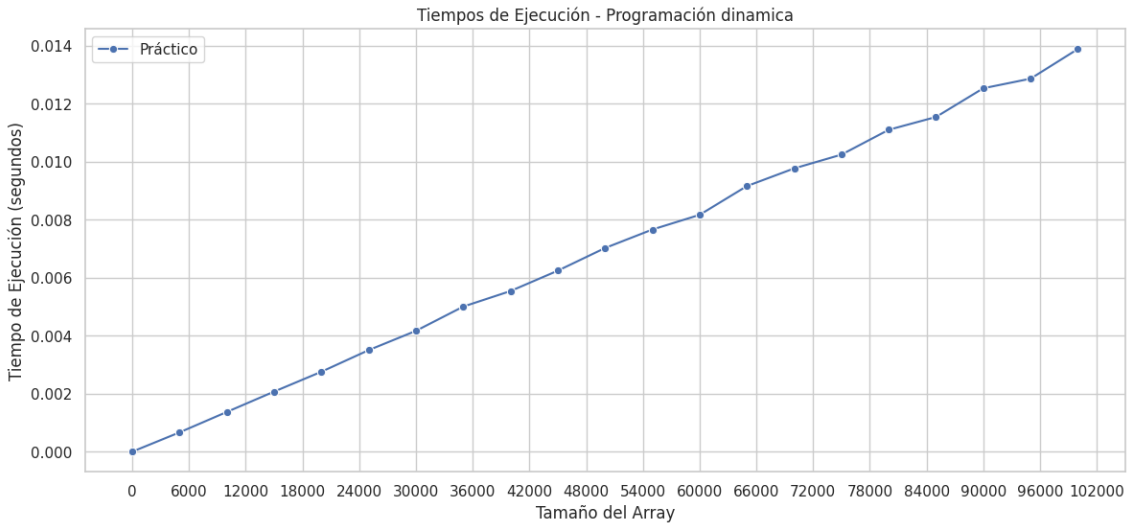
A la finalidad de reducir las variaciones debidas a factores externos —como la carga del sistema operativo, procesos en segundo plano o pequeñas fluctuaciones en el hardware— y obtener una curva más representativa de la complejidad algorítmica real, hemos iteramos múltiples veces (20) y calculamos el promedio del tiempo de ejecución para cada tamaño de entrada.

Datos obtenidos:

Tamaño Array	Tiempo de Ejecución (s)
0	0.000001
5,000	0.000621
10,000	0.001732
15,000	0.003132
20,000	0.002613
25,000	0.003101
30,000	0.003611
35,000	0.007671
40,000	0.004756
45,000	0.005519
50,000	0.006008
55,000	0.006929
60,000	0.007623
65,000	0.007881
70,000	0.008194
75,000	0.009160
80,000	0.009715
85,000	0.011269
90,000	0.010421
95,000	0.011771
100,000	0.011930

Cuadro 1: Resultados de tiempos de ejecución para distintos tamaños de array

Gráfico de datos obtenidos:

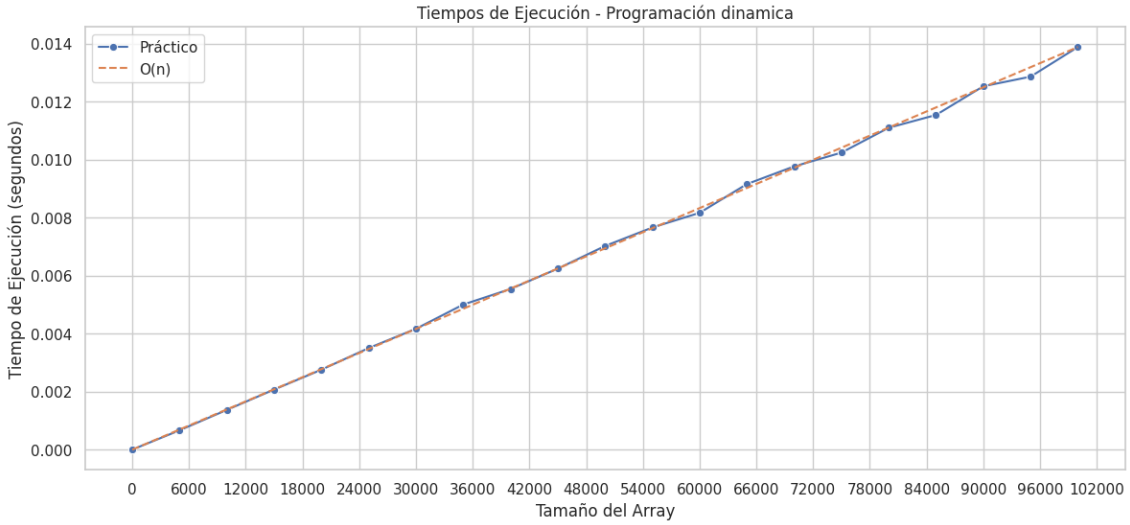


1.7. Informe de Resultados

1.7.1. Comparación de tiempos con la complejidad temporal teorica

Los tiempos de ejecución medidos para sets de datos de tamaños crecientes (0 a 100,000 elementos) muestran un crecimiento lineal, consistente con la complejidad $O(n)$. El gráfico incluye la curva de tiempos prácticos y la curva teórica $O(n)$, normalizada para coincidir con el primer punto. La curva práctica sigue de cerca la teórica permitiéndonos afirmar que se ha comportado linealmente para el set de datos utilizado.

Gráfico de datos obtenidos:



1.7.2. Comparación con el Problema 3

Los algoritmos de los Problemas 3 y 4 resuelven la misma situación: encontrar la subsecuencia contigua con suma máxima en un array de números enteros. Sin embargo, difieren significativamente en su enfoque y eficiencia.

Comparación de las complejidades computacionales

- **Problema 3 (Backtracking):** La complejidad temporal es $O(n^2)$, derivada de su búsqueda exhaustiva. Para cada posición inicial i, el algoritmo evalúa todas las subsecuencias que comienzan en i hasta el final del array, resultando en un número total de operaciones proporcional a la suma de los primeros n enteros, $\frac{n(n+1)}{2}$, que es del orden de $O(n^2)$.

- **Problema 4 (Programación Dinámica):** La complejidad temporal es $O(n)$, ya que realiza una sola iteración sobre el array, con operaciones constantes por elemento.

Teóricamente, el algoritmo de Programación Dinámica debería ser más eficiente, ya que su complejidad lineal $O(n)$ crece mucho más lentamente que la cuadrática $O(n^2)$ del Backtracking a medida que aumenta el tamaño del array. Esta diferencia se amplifica en arrays grandes, donde el costo computacional del Backtracking se vuelve prohibitivo.

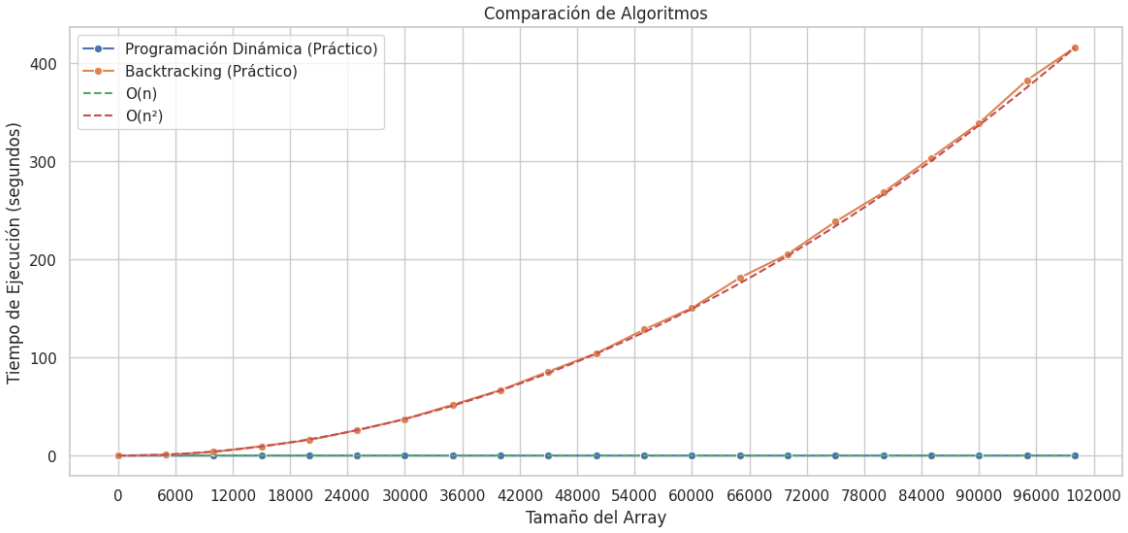
Para realizar la comparación del algoritmo desarrollado con estrategia de backtracking y el desarrollado con programación dinámica ampliamos las pruebas de backtracking hasta un array de 100.000 números.

Datos obtenidos:

Tamaño Array	Backtracking tiempos (s)	Programación dinámica tiempos (s)
0	0.000002	0.000001
5,000	0.957368	0.000621
10,000	3.749397	0.001732
15,000	9.613987	0.003132
20,000	15.884037	0.002613
25,000	26.083079	0.003101
30,000	37.190171	0.003611
35,000	51.901774	0.007671
40,000	66.804682	0.004756
45,000	85.791855	0.005519
50,000	104.298462	0.006008
55,000	128.547472	0.006929
60,000	150.525165	0.007623
65,000	181.409610	0.007881
70,000	205.418513	0.008194
75,000	238.576020	0.009160
80,000	268.388445	0.009715
85,000	303.800514	0.011269
90,000	339.048621	0.010421
95,000	382.862299	0.011771
100,000	415.985524	0.011930

Cuadro 2: Resultados de tiempos de ejecución para distintos tamaños de array

Gráfico de datos obtenidos:



La curva práctica del Backtracking sigue de cerca la teórica $O(n^2)$, mostrando un crecimiento exponencialmente más rápido que la curva lineal $O(n)$ de la Programación Dinámica. El gráfico confirma que, a partir de tamaños moderados (alrededor de 5000 elementos), la diferencia en rendimiento se vuelve significativa, haciendo del algoritmo de programación dinámica la elección preferida para arrays grandes.