UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA

TB024 - Teoría de Algoritmos | 2do Cuatrimestre 2025

Trabajo Práctico 1

Alumno	Padrón	Email
Mateo Bulnes	106211	mbulnes@fi.uba.ar
Maximiliano Nicolas Otero Silvera	108634	motero@fi.uba.ar
Emanuel Gaston Choque Ramirez	110895	egchoque@fi.uba.ar
Daniel Mamani	109932	dmamani@fi.uba.ar
Iñaki Pujato	109131	ipujato@fi.uba.ar

Índice

1. Problema 1

1.1. Enunciado

Tenemos un array A de tamaño n con enteros diferentes (positivos, negativos o cero), ordenado de menor a mayor. Diseñar e implementar un algoritmo de División y Conquista que determine el índice i tal que $A[i] = i$ o que indique si no existe tal i.

IMPORTANTE: el array ya viene ordenado. No incluir el ordenamiento en el análisis, en la determinación de la complejidad temporal ni en la medición de los tiempos de ejecución.

1.2. Análisis

1.2.1. Supuestos, condiciones y limitaciones

- El array esta compuesto de numeros enteros
- Los valores en el array son unicos
- La solucion es unica
- El array se encuentra ordenado

1.2.2. Casos base y combinación de resultados parciales

- Si el puntero 'izquierda' es mayor o igual a 'derecha' se retorna un -1 indicando que el algoritmo no encontro un indice que cumpla la condicion arr[i] == i
- El algoritmo divide el problema a la mitad en cada iteracion sin generar resultados parciales

1.3. Diseño

1.3.1. Pseudocódigo

```
FUNCION dc(arr: LISTA DE ENTEROS) RETORNA ENTERO
  RETORNAR run(arr, 0, LONGITUD(arr))
FIN FUNCION
  --
FUNCION run(arr: LISTA DE ENTEROS, izquierda: ENTERO, derecha: ENTERO) RETORNA ENTERO
  SI izquierda >= derecha ENTONCES
    RETORNAR -1
  FIN SI

  medio_idx: ENTERO = (izquierda + derecha) DIV 2

  SI arr[medio_idx] < medio_idx ENTONCES
    RETORNAR run(arr, medio_idx + 1, derecha)
  FIN SI

  SI arr[medio_idx] > medio_idx ENTONCES
    RETORNAR run(arr, izquierda, medio_idx)
  FIN SI

  RETORNAR medio_idx
FIN FUNCION
```

1.3.2. Estructuras de Datos

No se utilizan estructuras de datos auxiliares, unicamente el array provisto y dos punteros. Ya que unicamente estamos buscando el indice que cumpla la condicion arr[i] == i, por lo tanto podemos hacerlo sin modificar o copiar el array.

1.4. Seguimiento

1.4.1. Caso feliz

Sea el array [-1, 1, 3, 4, 5] de largo 5.

1.4.1.1 Iteracion 1

El algoritmo iniciara su primera iteracion con los argumentos

- arr: [-1, 1, 3, 4, 5]
- izquierda: 0
- derecha: 5

siendo $medio = (0 + 5)//2 = 2 => arr[medio] = 3 <> medio = 2$. Como $arr[medio] > medio$, movemos el puntero derecha hacia medio

1.4.1.2 Iteracion 2

El algoritmo iniciara su segunda iteracion con los argumentos

- arr: [-1, 1, 3, 4, 5]
- izquierda: 0
- derecha: 2

siendo $medio = (0 + 2)//2 = 1 => arr[medio] = 1 == medio = 1$. Como $arr[medio] = medio$, retornamos el resultado (1)

1.4.2. Caso triste

Sea el array [-1, 2, 3, 4, 5] de largo 5.

1.4.2.1 Iteracion 1

El algoritmo iniciara su primera iteracion con los argumentos

- arr: [-1, 2, 3, 4, 5]
- izquierda: 0
- derecha: 5

siendo $medio = (0 + 5)//2 = 2 => arr[medio] = 3 <> medio = 2$. Como $arr[medio] > medio$, movemos el puntero derecha hacia medio

1.4.2.2 Iteracion 2

El algoritmo iniciara su segunda iteracion con los argumentos

- arr: [-1, 2, 3, 4, 5]
- izquierda: 0
- derecha: 2

siendo $medio = (0 + 2)//2 = 1 => arr[medio] = 2 <> medio = 1$. Como $arr[medio] > medio$, movemos el puntero derecha hacia medio

1.4.2.3 Iteracion 3

El algoritmo iniciara su tercera iteracion con los argumentos

- arr: [-1, 2, 3, 4, 5]
- izquierda: 0
- derecha: 1

siendo $medio = (0 + 1)//2 = 0 => arr[medio] = -1 <> medio = 0$. Como $arr[medio] < medio$, movemos el puntero izquierda hacia medio mas uno

1.4.2.4 Iteracion 4

El algoritmo iniciara su cuarta iteracion con los argumentos

- arr: [-1, 2, 3, 4, 5]
- izquierda: 1
- derecha: 1

como izquierda >= derecha, el algoritmo retorna (-1) indicando que no encontro un indice que cumpla arr[i] == i

1.5. Complejidad

El algoritmo esta realizando 1 llamado recursivo (A = 1), partiendo a la mitad el problema en cada llamado recursivo (B = 2), luego, todo lo que no son llamados recursivos se esta ejecutando en $O(1)$ (C = 0)
Por lo tanto, la ecuacion de recurrencia para este algoritmo es $T(n) = AT(n/B) + O(n^C) = T(n/2) + O(1)$.
Utilizando el teorema maestro, tenemos que la complejidad temporal del algoritmo es
$log_B(A) = log_2(1) = 0 = C => T(n) = O(log_2(n))$

1.6. Sets de Datos

Para los sets de datos, se definieron algunos casos base y luego se generan en runtime mediante una funcion que agrega *samplesize* arrays compuestos de numeros aleatorios hasta un largo N, siendo N un numero aleatorio entre *max* y *min*, y se asegura que este array cumpla con las condiciones y limitaciones del algoritmo. Se utilizo la seed 777 tanto para la generacion de set de datos para las pruebas como para medir el tiempo de ejecucion del algoritmo.

Dado que el algoritmo de fuerza bruta que resuelve este problema es relativamente sencillo, para las pruebas se hace un doble-check esperando que el resultado que el algoritmo de DyC retorna sea el mismo que el calculado 'a mano' y por fuerza bruta.

1.7. Tiempos de Ejecución

Para los tiempos de ejecucion se tuvo que agregar un for que cuente hasta 50 millones para agregar tiempo de procesamiento entre cada iteracion, y que el tiempo de ejecucion resultante no sea puramente interrupciones del sistema. Esto debido a que el algoritmo al ser teoricamente $O(log(n))$ los tiempos de ejecucion con arrays de por ejemplo, 100 millones de numeros tardaban 30 - 40 microsegundos incluso corriendo ese set de datos en un docker con el flag $--cpus = 0{,}01$ siendo la minima cantidad de procesador que se le puede otorgar al container.

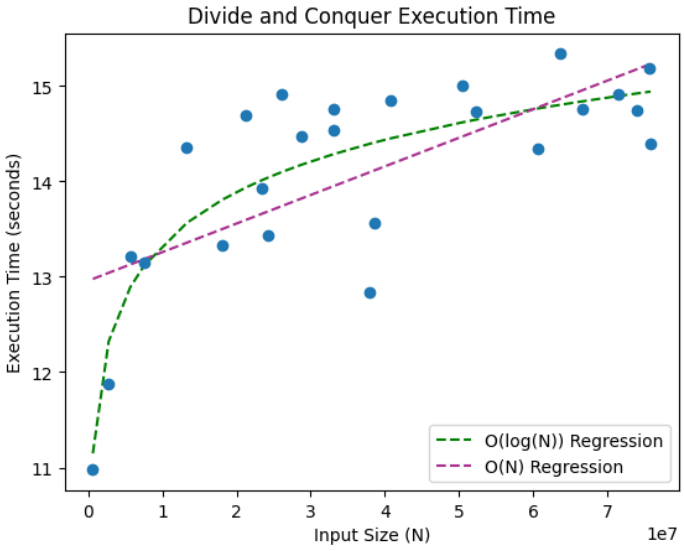

Figura 1: Comparacion de tiempos de ejecucion segun el largo del array

1.8. Informe de Resultados

Como se puede ver en la figura 1 los tiempos de ejecucion se ajustan mejor a una complejidad temporal de $O(log(n))$ como lo habiamos planteado teoricamente que a una $O(n)$, incluso habiendo incluido el for previamente mencionado ya que el mismo lo podemos considerar un tiempo extra "fijo" añadido entre cada iteracion por lo cual a medida que n crece, se requieren mas iteraciones por ende se llamara mas veces al for.

2. Problema 2

2.1. Enunciado

Tenemos otro array A de tamaño n, como el del Problema 1, pero que no está ordenado. Llamaremos intervalo positivo al subarray A[i..j] (ij) tal que la suma de sus valores es mayor (estricto) que cero. Diseñar e implementar un algoritmo Greedy que calcule la mínima cantidad de intervalos positivos en todo el array A. Por ejemplo, si el array fuera el siguiente, el algoritmo debería devolver 3 (los intervalos positivos se identifican en gris):

+3	-5	+7	-4	+1	-8	+3	-7	+5	-9	+5	-2	+4

2.2. Análisis

2.2.1. Supuestos, condiciones y limitaciones

- Array compuesto de números enteros desordenados con al menos 1 elemento
- Retorna la cantidad de intervalos positivos que tengan mas de 1 elemento
- No requiere bibliotecas de ningun tipo
- La solución es única y óptima
- Contigua significa que los elementos elegidos deben estar adyacentes en el array original, sin omitir ninguno en el medio.
- No se permite reordenar elementos

2.2.2. Regla Greedy

La regla greedy esta compuesta de dos partes. La primera e inmediata es revisar si la suma acumulada es positiva. La otra es revisar si al convertirse en negativo si en algún momento volverá a ser positivo. Este análisis es muy importante ya que sigue siendo greedy priorizando su solución local inmediata, sin tomar una decisión errónea mas adelante.

La primera parte de la regla nos permite tomar una decisión inmediata y local de que si la suma es positiva sigo siendo parte del intervalo positivo y la incluyo continuando el flujo. Y la segunda parte nos previene de tomar decisiones incorrectas respecto al subconjunto positivo y su tamaño.

2.2.3. Análisis solución óptima

Este algoritmo devuelve la solución óptima siempre. Para ello toma en cada paso un par de chequeos para ver que la conclusión a la que está llegando es la deseada.

Primero revisa si la suma acumulada es positiva y en caso de serlo añade el item al subintervalo que esta considerando, en caso de que no hubiera tal subintervalo, restablece la sumatoria al valor de numero si este es positivo. Luego, en caso de que la suma se transforme negativa, revisa si en algún momento más adelante la suma volverá a ser positiva. Esto es muy importante ya que si no se hace se puede dar un caso como [1, -2, -1, -3, -4, 15] en el cual la respuesta es 1, pero si uno solo toma conclusiones locales dará 0. No solo eso, es fundamental mirar hacia adelante cuando hace falta para lograr soluciones validas, por ejemplo al hacer [-10, -2, -1, 5] la minima cantidad de subintervalos es 1 ya que [-2, -1, 5] suma 2. Del mismo modo, en casos como [1, 3, 5, 1, 4] se mantiene solo el uso de la decisión inmediata ya que nunca debe revisar adelante, optimizando al máximo la decisión local.

Al combinar ambas metodologías podemos asegurar que siempre obtendremos la solución óptima a nuestro problema.

2.3. Diseño

2.3.1. Pseudocódigo

```
FUNCION sera_positivo(arr: LISTA, i: ENTERO, suma: ENTERO) RETORNA BOOL
  PARA j DESDE i+1 HASTA fin(arr)
    suma := suma + arr[j]
    SI suma > 0 ENTONCES RETORNAR VERDADERO
  FIN PARA
  RETORNAR FALSO
FIN FUNCION

FUNCION greedy(arr: LISTA) RETORNA ENTERO

  PARA cada elemento x EN arr
    suma INCREMENTA x

    SI suma > 0 ENTONCES
      AGREGAR x A subintervalo
    SINO SI x < 0 Y sera_positivo(arr, pos(x), suma) ENTONCES
      AGREGAR x A subintervalo
    SINO SI x > 0 Y subintervalo VACIO ENTONCES
      INICIAR subintervalo CON x
      suma IGUAL x
    SINO SI x < 0 Y NO sera_positivo(arr, pos(x), suma) ENTONCES
      CERRAR subintervalo (SI longitud > 1 INCREMENTAR intervalos)
      REINICIAR suma Y subintervalo
    FIN SI
  FIN PARA

  SI subintervalo NO VACIO Y longitud > 1 ENTONCES
    INCREMENTAR intervalos
  FIN SI

  RETORNAR intervalos
FIN FUNCION
```

2.3.2. Estructuras de Datos

La unica estructura que usamos (ademas del array inherente al enunciado) es otro array que contiene el subconjunto que esta siendo formado con la suma parcial positiva. Usamos un array ya que permite con sus primitivas saber rapidamente si el mismo es de tamaño mayor a 1 elemento o si esta vacio.

2.4. Seguimiento

Para hacer el seguimiento vamos a usar el array [2, 3, -5, -2, 1]:

1. Comienza revisando el 2, suma el numero a la suma parcial y por ser el 2 positivo, lo agrega al subconjunto que se está considerando. Como estan todas las condiciones dadas, continua la iteracion.

   Arr: [**2**, 3, -5, -2, 1]

   Subconjunto: [2]

   Suma: 2

Intervalos: 0

2. Luego pasa al 3. Nuevamente lo suma y al ser positivo lo agrega al subconjunto. Hasta acá ambos son los casos mas amigables para la regla greedy, por lo que la solucion es inmediata.

   Arr: [2, **3**, -5, -2, 1]

   Subconjunto: [2, 3]

   Suma: 5

   Intervalos: 0

3. Luego toca -5. Al realizar la suma la misma queda en 0, en este caso el subintervalo deja de ser positivo y comienza a revisar que sucederá mas adelante. Entra en la funcion *turns_positive* y revisa que sucederá mas adelante. En este caso vemos que la suma restante mantendrá el no positivo (0 - 2 + 1 = -1) por lo que corta al subconjunto en el estado en el que esta y reincia la suma. Revisa que el subconjunto que termino sea mayor a uno para cumplir que la suma contenga mas de un elemento, y termina este paso.

   Arr: [2, 3, **-5**, -2, 1]

   Subconjunto: [ ]

   Suma: 0

   Intervalos: 1

4. A continuacion el -2. Suma el numero y al ser este negativo y no volverse positivo en el futuro continua.

   Arr: [2, 3, -5, **-2**, 1]

   Subconjunto: [ ]

   Suma: 0

   Intervalos: 1

5. Por ultimo llega al 1. Suma y queda positivo por lo que lo incluye en el subconjunto. En este momento se termina el recorrido por lo que se revisa si hay items en el subconjunto. El problema es que si bien hay, hay solo 1, por lo que no se suma un intervalo adicional.

   Arr: [2, 3, -5, -2, **1**]

   Subconjunto: [1]

   Suma: 1

   Intervalos: 1

2.5. Complejidad

El algoritmo propuesto tiene complejidad de $O(N^2)$. Esto se debe a que el bucle principal recorre todo el algoritmo siendo O(N) y en caso de requerirlo recorre hacia adelante para revisar que sea positivo. La recorrida hacia adelante siempre sera algo menor a N, ya que parte desde el item actual, pero en la notación BigO esto es igualmente $O(N^2)$.

En ambas recorridas las tareas que se realizan son O(1) ya que son solo chequeos, pero cuando se anidan ambas tienen complejidad O(N) por lo que dan el resultante $O(N^2)$.

Si bien la complejidad no es la mejor, es la optima debido a que el enunciado no permite ordenar el algoritmo, lo cual rompería los subconjuntos, por lo que cualquier revision respecto a la potencial positividad de la suma implicará $O(N^2)$.

2.6. Sets de Datos

Para los Sets de Datos en los test se generaron casos manuales. El problema con la generacion automatica es que la unica forma de determinar que el funcionamiento es correcto es con supervision humana para contar los subconjuntos y monitorear el funcionamiento. Se generaron 14 casos, buscando incluir multiples escenarios posibles:

- Casos con 1 solo subconjunto positivo
- Casos sin subconjuntos positivos
- Casos de todos items positivos
- Casos con multiples subconjuntos positivos

En los tres escenarios se busco realizar diferencias de ordenamiento que fuerzen al algoritmo a cubrir los casos borde. Por ejemplo, es diferente el uso de la regla Greedy si el array es [-2, 1, 2] que si fuese [2, 1, -2].

Para las mediciones de ejecución si se generaron Casos automaticos para poder crear subconjuntos mayores. Haremos pruebas con multiples tamaños y varias corridas para cada uno y usaremos el promedio de las corridas para revisar nuestros resultados.

2.7. Tiempos de Ejecución

Para medir los tiempos de ejecucion se genero de manera aleatoria usando la misma seed (777) arrays de tamaño N, con enteros entre 100 y -100. Se midio N = [10, 100, 1000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000] midiendo 10 corridas y usando el promedio. Los resultados obtenidos fueron:

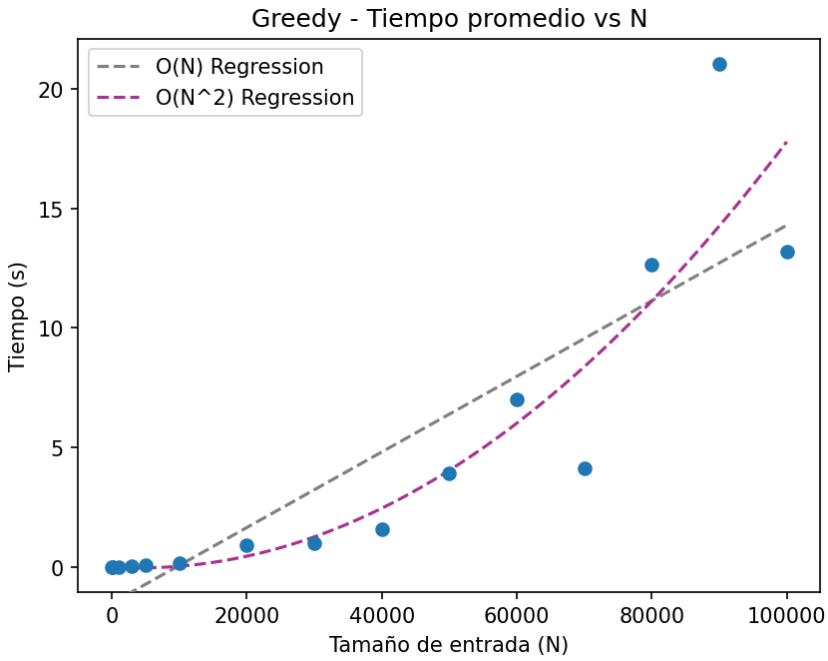

Figura 2: Resultados de ejecucion promedio.

N	Promedio (s)
10	0.00000776
100	0.00003589
1000	0.00313651
3000	0.02539459
5000	0.06251066
10000	0.18118074
20000	0.89591365
30000	1.00299926
40000	1.57790497
50000	3.93786702
60000	7.02210779
70000	4.12812427
80000	12.64282358
90000	21.04938847
100000	13.20983805

Cuadro 1: Tiempos de ejecucion del algoritmo Greedy (10 ejecuciones por tamaño).

2.8. Informe de Resultados

Vemos en nuestros resultados que efectivamente al hacer varias corridas sigue nuestra estimacion de complejidad. Algo muy importante a tener en cuenta es que en algunos casos puede darse una ejecucion lineal y nunca entrar al subcaso de revision futura. Esto condiciona fuertemente el tiempo, por lo que si uno corre pocas veces el algorimo puede obtener una gran dispersion que no se asemeje a la complejidad estimada.

3. Problema 3

3.1. Enunciado

Tenemos un array A de números enteros (positivos, negativos o cero) como el del Problema 2, no ordenado. Queremos encontrar la secuencia de uno o más elementos contiguos que sumen el máximo valor posible. Desarrollar un algoritmo de Backtracking para encontrar dicha secuencia

3.2. Análisis

3.2.1. Supuestos, condiciones y limitaciones

- Trabajamos con un arreglo de n números enteros (positivos, negativos o ceros) desordenados, con al menos 1 elemento (subarreglos no vacíos).
- Si todos los números son negativos, la subsecuencia contigua de suma máxima será aquel elemento menos negativo (el de mayor valor) por sí solo.
- Si existen números positivos, la suma máxima será no negativo y potencialmente mayor si se combinan con otros valores.
- Contigua significa que los elementos elegidos deben estar adyacentes en el array original, sin omitir ninguno en el medio.
- No se permite reordenar elementos ni seleccionar subsecuencias no contiguas.

3.2.2. Análisis de Podas

Durante la exploración Backtracking, se incorporan condiciones de poda para evitar explorar ramas que no conducirán a soluciones óptimas. En particular, si al extender una subsecuencia contigua la suma parcial se vuelve negativa o cero, el algoritmo poda esa rama. La razón es que una subsecuencia cuyo prefijo acumula suma negativa no puede ser parte de la solución óptima.

Por ejemplo, si tenemos un index que suma -5 seguido de un 10, es mejor ignorar el -5 y empezar la subsecuencia en 10 directamente. Esta poda reduce significativamente las ramas a explorar sin omitir soluciones factibles.

3.2.3. Análisis del tamaño de soluciones

El espacio de soluciones factibles (todas las subsecuencias contiguas posibles) crece en el orden de $O(n^2)$. Para un arreglo de tamaño n existe un número combinatorio de subarreglos contiguos posible dad opor la fórmula $\frac{n \cdot (n+1)}{2}$ (*GeeksforGeeks*). Esto incluye todas las subsecuencias de longitud 1 (hay n de las mismas), de longintud 2 (hay $n-1$) y asi gasta la de longitud n (1 sola, el arreglo completo).

Por ejemplo un array de $n = 5$ elementos tienen $\frac{5 \cdot 6}{2} = 15$ subarrays contiguos distintos. Entonces el algoritmo de Backtracking, en el peor de los casos, podría tener que evaluar la suma de cada una de esas subsecuencias para encontrar la máxima. Gracias a las podas mencionadas previamente, muchas de estas combinaciones se descartan antes de recorrerlas por completo, pero en el peor de los casos igualmente se explorarían casi todas las posibilidades.

En conclusión, el espacio de búsqueda es cuadrático en función de n.

3.3. Diseño

3.3.1. Pseudocódigo

```
Funcion max_subarray_backtrack(arr: Lista de enteros):
    best_sum = -infinito            // Mejor suma encontrada
    best_start = 0, best_end = 0    // Indices del subarray de mejor suma

    Funcion extend_subarray(start_index, current_index, current_sum):
        Si current_index >= len(arr) Entonces
            Retornar
        current_sum = current_sum + arr[current_index]

        Si current_sum > best_sum Entonces  // Es mejor suma que la que teniamos
    guardada
            best_sum = current_sum
            best_start = start_index
            best_end = current_index

        Si current_sum > 0 Entonces
            # Solo extendemos si la suma parcial sigue siendo positiva
            extend_subarray(start_index, current_index + 1, current_sum)

        // Si current_sum se vuelve negativa, no extendemos mas la subsecuencia (
    PODA)
        retornar

    Funcion backtrack_from(index):
        Si index == len(arr) Entonces
            Retornar

        extend_subarray(index, index, 0)
        backtrack_from(index + 1)

    backtrack_from(0)                           // Llamada inicial
    Retornar best_sum, (best_start, best_end)   // Retorna la mejor suma y sus
     ndices
```

3.3.2. Estructuras de Datos

Para el algoritmo se utilizan las siguientes Estructuras de Datos:

- Arreglo de entrada (arr): Estructura base que contiene los números enteros.
- Variables para el seguimiento (best_sum, best_start, best_end): Se utilizan para llevar seguimiento de la mejor solución.
- Pila de ejecución (implícita con la recursión): La solución aprovecha la recursividad para explorar las opciones. No se utiliza una estructura de pila explícitamente, pero la propia call stack del programa actúa coo tal en el backtracking, almacenando el estado de current_index y current_sum en cada nivel de profundidad mientras extiende subsecuencias.

3.4. Seguimiento

Para mostrar el funcionamiento, consideramos el array de ejemplo: [2, -4, 3, -1, 2]. A continuación, seguimos el algoritmo paso a paso, mostrando las decisiones y podas:

Inicio en índice 0: Comenzamos una subsecuencia en la posición 0:

- Incluimos $arr[0] = 2$. Suma parcial es 2.
  Mejor suma encontrada = 2 (subarray [2]).
- La suma parcial (2) es positiva, por lo que extendemos al siguiente índice.

- incluimos $arr[1] = -4$. Suma parcial es $2 + (-4) = -2$. ‘
  Mejor suma sigue siendo 2 (la subsecuencia [2] aún es mejor).

- Ahora la suma parcial se volvió negativa (-2).
  Poda: Dado que continuar esta subsecuencia solo añadiría más elementos a un total negativo (que no superará a 2), el algoritmo detiene la extensión de la subsecuencia iniciada en 0.
  No se consideran [2, -4, 3] ni más largas a partir de este inicio, porque ya el prefijo [2, -4] arruinó la suma.
  Se descarta seguir a [-4, 3], [-4, 3, -1], etc.

Inicio en índice 1: Comenzamos una subsecuencia nueva en la posición 1:

- Incluimos $arr[1] = -4$. Suma parcial es -4.
  Mejor suma sigue siendo 2 (la subsecuencia [2] sigue siendo la mejor hasta ahora, ya que $-4 < 2$

- La suma parcial es negativa desde el principio.
  Poda inmediata: No tiene sentido extender [-4] con elementos posteriores.

Inicio en índice 2: Nueva subsecuencia comenzando en 2:

- Incluimos $arr[2] = 3$. Suma parcial es 3.
  Mejor suma encontrada = 3 (subarray [3] supera al anterior 2).

- Suma parcial positiva (3), entonces extendemos.

- Incluimos $arr[3] = -1$. Suma parcial es $3 + (-1) = 2$.
  Mejor suma permanece 3 (la subsecuencia [3] aún es mejor, porque $2 < 3$).

- La suma parcial tras incluir -1 es 2 (positiva). Continuamos extendiendo.

- Incluimos $arr[4] = 2$. Suma parcial es $2 + 2 = 4$.
  Mejor suma encontrada = 4 ahora (la subsecuencia [3, -1, 2] tiene suma 4).

- Hemos llegado al final del array desde este inicio. La subsecuencia contigua [3, -1, 2] completa tiene suma 4, ques la mayora hasta el momento.

Inicio en índice 3: Comenzamos subscecuencia en 3:

- Incluimos $arr[3] = -1$. Suma parcial es -1.
  Mejor suma sigue siendo 4 (ningún cambio, $-1 < 4$).

- Suma parcial negativa.
  Poda: Se descarta extender [-1] con el siguiente elemento, ya que cualquier subsecuencia que empiece en -1 sería empeorada por este prefijo negativo.

Inicio en índice 4: Nueva subsecuencia en la última posición:

- Incluimos $arr[4] = 2$. Suma parcial es 2.
  Mejor suma sigue siendo 4 (la subsecuencia [3, -1, 2] sumó 4, mayor que 2).

- No hay más elementos para extender. Se cierra el recorrido.

Tras evaluar todas las posibilidades, el algoritmo concluye que la subsecuencia contigua de suma máxima es [3, -1, 2] con suma 4. Se puede destacar cómo las podas evitaron revisar muchas subsecuencias que no nos hubieran aportado ningun valor.

3.5. Complejidad

El algoritmos de backtracking utilizado tiene, en el peor de los casos, un tiempo de ejecución de $O(n^2)$. Esto se debe a que, en el peor escenario no tenemos podas (Por ejemplo, si todos los números mantienen la suma parcial positiva al acumularse), la recursión interna calculará la suma de todas las posibles subsecuencias contiguas (*AfterAcademy*). En cada extensión de subsecuencia, la suma se actualiza en $O(1)$, de modo que considerar cada subarray es constante tras la acumulación incremental. Por lo tanto recorrer $\frac{1}{2} \cdot n^2$ subarrays con operaciones constantes lleva un orden $O(n^2)$ global.

Las podas implementadas en el algoritmo tienden a mejorar el caso promedio (por ejemplo, cortando ramas cuando hay sumas negativas), pero no alteran la cota de peor caso.

Un punto a destacar es que si no utilizáramos la optimización de acumular la suma parcialmente, un enfoque totalmente ingenuo tardaría $O(n^3)$. Nuestro algoritmo evita recalcular sumas desde cero usando la técnica incremental en la recursión, reduciendo asi la complejidad a $O(n^2)$.

3.6. Sets de Datos

Se proponen varios conjuntos de datos manualmente diseñados para validar el algoritmo en distintos escenarios. Para cada set se indica el array de entrada y resultado esperado (suma máxima y la subsecuencia contigua que logra esa suma)

- **Set 1 - Mixto con positivos y negativos:** [2, -4, 3, -1, 2]
  Resultado Esperado: Suma máxima = 4, obtenida por la subsecuencia contigua [3, -1, 2].
  Explicación: En este arreglo, la mejor subsecuencia comienza en el tercer elemento. Aunque 2 al inicio aporta suma positiva, se ve reducida por el -4, y la mejor opción termina siendo arrancar en 3.
- **Set 2 - Todos números negativos:** [-3, -1, -7]
  Resultado Esperado: Suma máxima = -1, obtenida por la subsecuencia contigua [-1].
  Explicación: Cuando todos los elementos del arreglo son negativos, la mayor suma posible es tomar el número menos negativo (el de mayor valor). En este caso, -1 es mayor que -3 y -7, por lo que la subsecuencia de suma máxima es simplemente [-1].
- **Set 3 - Todos números positivos:** [1, 2, 3, 4]
  Resultado Esperado: Suma máxima = 10, obtenida por la subsecuencia contigua [1, 2, 3, 4].
  Explicación: Si no hay números negativos que penalicen la suma, la mejor estrategia es tomar todos los elementos del arreglo.
- **Set 4 - Mezcla con cero incluidos:** [0, -1, 3, -2, 0, 4]
  Resultado Esperado: Suma máxima = 5, obtenida por la subsecuencia contigua [3, -2, 0, 4].
  Explicación: En este caso, el subarray óptimo empieza en el valor 3 (índice 2). La suma se desarrolla como: $3 + (-2) = 1$, luego $1 + 0 = 1$, y $1 + 4 = 5$. El cero no afecta la suma pero permite continuar la contigüidad hasta llegar al 4.

3.7. Tiempos de Ejecución

Los sets de datos mencionados en el punto anterior se ejecutaron y obtuvimos los siguientes resultados:

- **Set 1:** $10,33$ ms
- **Set 2:** $2,88$ ms

- **Set 3:** $3,79$ ms
- **Set 4:** $3,5$ ms

Los mismos resultados fueron plasmados en el siguiente gráfico:

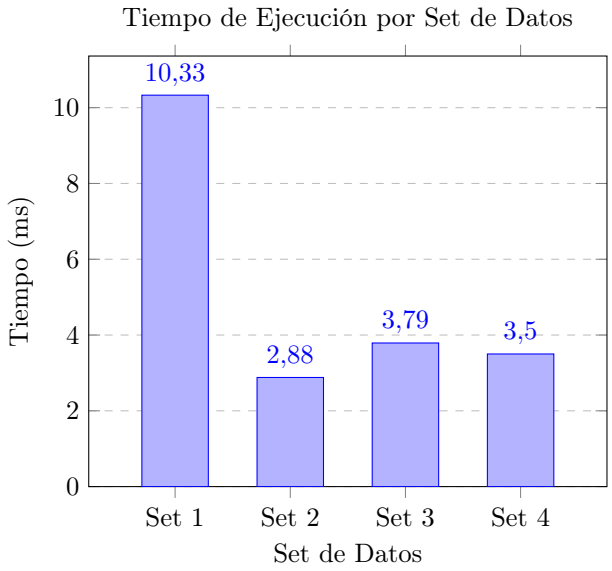

Figura 3: Comparación de los tiempos de ejecución del algoritmo para 4 sets de datos.

3.8. Informe de Resultados

El algoritmo tiene complejidad $O(n^2)$ en el peor caso. En las mediciones, los tiempos no siguen un crecimiento cuadrático por dos motivos:

- Tenemos podas que acortan ramas en instancias con negativos/ceros.
- Encontramos ruido propio de mediciones cortas en Python.

Para visualizar la relación, se superpone una curva de referencia $k * n^2$ ajustada a los datos. Aún con dispersión, la comparación es consistente con un crecimiento subcuadrático observado en estos casos pequeños, y con la cota cuadrática teórica cuando el patrón de datos fuerza el peor caso.

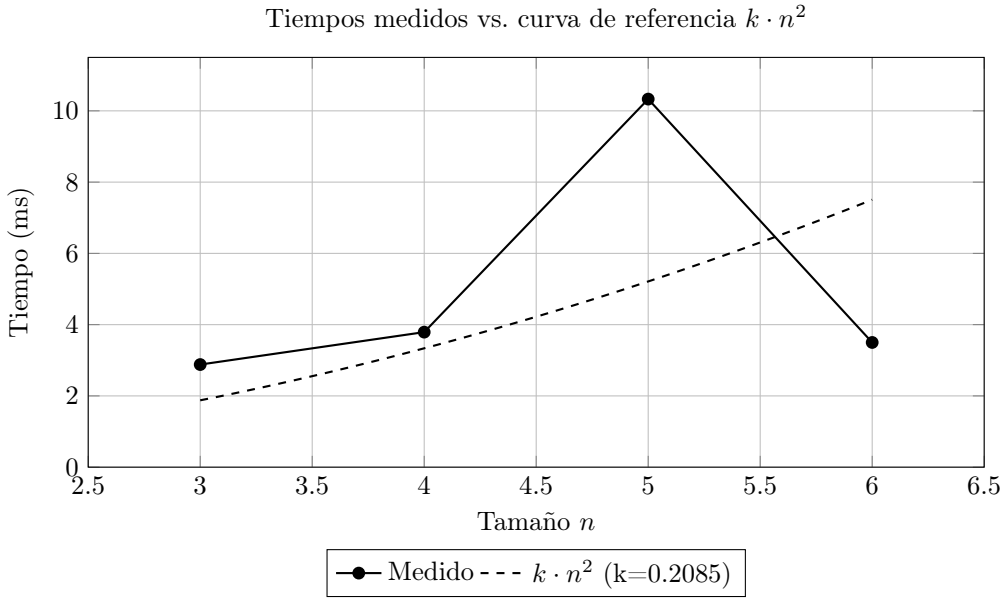

Figura 4: Comparación entre tiempos medidos y la curva de referencia cuadrática. Las diferencias obedecen a podas (caso no-peor) y ruido de medición en tamaños pequeños.

4. Problema 4

4.1. Enunciado

Para el Problema 3, desarrollar un algoritmo de Programación Dinámica que encuentre la secuencia que sume el máximo valor posible.

4.2. Análisis

4.2.1. Supuestos, condiciones y limitaciones

- El array de entrada está compuesto únicamente por números enteros (positivos, negativos o cero). No contiene letras ni caracteres especiales.
- Siempre existe una subsección de uno o más elementos cuya suma es máxima, ya que la subsección puede ser un único elemento.
- La solución debe ser una subsección contigua de uno o más elementos.
- Siempre existe una subsección con la suma máxima, incluso si la suma es negativa.
- El ambiente de la ejecución evita el desbordamiento (overflow) de enteros en las variables de suma.

4.2.2. Ecuación de Recurrencia

La solución óptima para el problema se basa en la siguiente ecuación de recurrencia:

$$S[i] = max(S[i-1] + A[i], A[i]) \tag{1}$$

Caso Base:

$$S[0] = A[0] \tag{2}$$

La suma máxima que termina en el primer índice es el primer elemento.

Explicación y Fundamentación:
Para resolver este problema con Programación Dinámica se definió el siguiente subproblema:
Encontrar la suma máxima de una subsección contigua que termina en el índice i. Esta suma se la va a llamar como S[i].
De esta forma, el cálculo de S[i] se realiza tomando una decisión respecto al elemento actual A[i]:

- Opción 1: Ignorar todas las subsecciones anteriores y considerar que la subsección máxima que termina en i es simplemente el elemento A[i] por sí solo. Esto ocurre si la suma de la subsección que terminaba en i-1 era negativa, lo que significa que extenderla solo empeoraría el resultado.
- Opción 2: Extender la subsección óptima que terminaba en el índice i-1 al sumarle el elemento A[i]. Esta opción es conveniente si la suma S[i-1] era positiva, ya que contribuirá a un valor más alto.

La ecuación de recurrencia planteada elige la mejor de estas 2 opciones. La solución general para todo el array es el valor máximo para todo el array S.

4.2.3. Análisis de Requisitos

El algoritmo diseñado usando programación dinámica para encontrar la subsección de máxima suma cumple con los 2 requisitos fundamentales para esta técnica: la **Subestructura Óptima** y los **Subproblemas Superpuestos**.

- **Subestructura Óptima:** La solución óptima al problema principal (suma máxima en todo el array) se construye a partir de las soluciones óptimas de sus subproblemas. En este caso, la solución final es el valor máximo entre todas las sumas máximas de subsecciones que terminan en cada posición del array. La ecuación de recurrencia definida, S[i]=max(S[i-1]+A[i],A[i]), garantiza que para cada índice i, se encuentra la suma máxima óptima que termina en esa posición, tomando como base la solución óptima del subproblema anterior
- **Subproblemas Superpuestos:** Este principio se cumple porque, al resolver el problema de forma iterativa, el cálculo de la suma máxima que termina en la posición i requiere la suma máxima que termina en la posición i-1. A medida que se avanza a través del array, se resuelve una serie de subproblemas interdependientes.

4.2.4. Uso de Memoization

Para evitar recalcular valores ya obtenidos, se almacenan los resultados de cada subproblema en un array auxiliar S. De esta forma, cada subproblema S[i] se calcula **una sola vez**, y luego se reutiliza para obtener S[i+1]. Esta técnica impide la redundancia de cálculos que aparecería en un enfoque recursivo sin memoization, y es lo que me permite que el algoritmo de programación dinámica sea mucho más eficiente que una solución de fuerza bruta o backtracking.

4.3. Diseño

4.3.1. Pseudocódigo

```
Funcion subsecuencia_max(A):
    asigno la longitud del array A a la variable n
    Si n = 0:
        devuelve cero y no hay subseccion
    suma_maxima = A[0]
```

```
        inicio = 0
        fin = 0
        suma_actual = A[0]
        inicio_temp = 0

        Para cada i desde 1 hasta n-1:
            Si suma_actual + A[i] > A[i]:
                suma_actual += A[i]
            Sino:
                suma_actual = A[i]
                inicio_temp = i

            Si suma_actual > suma_maxima:
                suma_maxima = suma_actual
                inicio = inicio_temp
                fin = i
        devuelve la suma max con los indices de inicio y fin de la
    subseccion
```

4.3.2. Estructuras de Datos

No se utilizan estructuras de datos auxiliares, sólo variables auxiliares para almacenar la suma actual y los índices de inicio y fin de la subsección cuya suma es máxima. La elección de no utilizar un array auxiliar para almacenar todas las sumas S[i] se justifica por la naturaleza lineal de la dependencia en la ecuación de recurrencia. Dado que S[i] sólo depende de S[i-1] y no de valores anteriores, se puede reutilizar una única variable que almacene la suma actual en cada paso. Esto permite una complejidad espacial O(1), maximizando la eficiencia.

4.4. Seguimiento

Para demostrar el funcionamiento del algoritmo, se hará un ejemplo de seguimiento usando el siguiente arreglo:

$$A = [3, -5, 7, 1, -8, 3, -7, 5] \tag{3}$$

Para este ejemplo, se seguirán los pasos del algoritmo para calcular el array de sumas óptimas, S, y al mismo tiempo se rastrearán las variables para la suma máxima global y sus índices.

- **suma actual:** La suma máxima de la subsección que termina en la posición actual.
- **suma maxima:** La suma máxima encontrada hasta el momento.
- **inicio:** El índice de inicio de la subsección con la suma máxima total.
- **fin:** El índice de fin de la subsección con la suma máxima total.
- **inicio temp:** Un índice auxiliar para rastrear el inicio de la subsección actual.

i	A[i]	suma_ant	suma_actual = max(suma_ant + A[i], A[i])	suma_max	inicio	fin	inicio_temp
-	-	-	-	$-\infty$	-	-	0
0	3	-	$3 = \max(-\infty + 3, 3)$	3	0	0	0
1	-5	3	-2 = max(3 + (-5), -5)	3	0	0	0
2	7	-2	7 = max(-2 + 7, 7)	7	2	2	2
3	1	7	8 = max(7 + 1, 1)	8	2	3	2
4	-8	8	0 = max(8 + (-8), -8)	8	2	3	2
5	3	0	3 = max(0 + 3, 3)	8	2	3	5
6	-7	3	-4 = max(3 + (-7), -7)	8	2	3	5
7	5	-4	5 = max(-4 + 5, 5)	8	2	3	7

Al finalizar el recorrido, la suma_maxima es 8, con los índices de inicio y fin de 2 y 3, respectivamente. La subsección con la suma máxima es A[2...3], que corresponde a los elementos [7,1].

4.5. Complejidad

El algoritmo diseñado para encontrar la subsección contigua con la suma máxima posee una complejidad temporal lineal, **O(n)**.

Justificación:
La complejidad se determina analizando la cantidad de operaciones que realiza el algoritmo en función del tamaño del array de entrada, n:

1. **Inicialización:** Al comienzo de la función, se realiza un número constante de asignaciones a variables (como suma_maxima, inicio, fin, etc). Esta fase tiene una complejidad O(1).
2. **Bucle Principal:** La mayor parte del trabajo se realiza dentro de un bucle que itera sobre el array A. El bucle se ejecuta exactamente n-1 veces (desde i=1 hasta n-1).
3. **Operaciones por iteración:** Dentro del bucle, el algoritmo realiza un número fijo y constante de operaciones en cada iteración:
   - Una suma (suma_actual + A[i])
   - Una o dos comparaciones para aplicar la ecuación de recurrencia
   - Una o dos asignaciones para actualizar la suma actual y los índices temporales.
   - Una o dos comparaciones adicionales para actualizar la solución global (suma_maxima, inicio, fin)

Dado que el número de operaciones por iteración es constante (k), el tiempo total empleado por el bucle es proporcional a k * (n-1). En términos de notación Big O, las constantes y los términos de orden inferior se descartan, lo que resulta en una complejidad final de **O(n)**.
Este rendimiento de O(n) es el óptimo para este problema, ya que es imposible resolverlo sin, al menos, examinar cada elemento del array una vez. La programación dinámica logra esta eficiencia al evitar la redundancia de cálculos, característica que lo hace superior al algoritmo de Backtracking del problema 3, que tiene una complejidad exponencial en el peor de los casos.

4.6. Sets de Datos

Para validar el algoritmo, se utilizaron los mismos conjuntos de datos definidos en el Problema 3 ya que permiten cubrir distintos escenarios.

- **Set 1 - Mixto con positivos y negativos**
  Entrada: [2, -4, 3, -1, 2]
  Resultado esperado: 4, subsecuencia [3, -1, 2]
- **Set 2 - Todos números negativos**
  Entrada: [-3, -1, -7]
  Resultado esperado: -1, subsecuencia [-1]
- **Set 3 - Todos numeros positivos**
  Entrada [1, 2, 3, 4]
  Resultado esperado: 10, subsecuencia [1, 2, 3, 4]
- **Set 4 - Mezcla con ceros**
  Entrada: [0, -1, 3, -2, 0, -4]
  Resultado esperado: 5, subsecuencia [3, -2, 0, 4]

4.7. Tiempos de Ejecución

Para los tiempos de ejecución se utilizaron los sets de datos mencionados teniendo así los siguientes resultados:

- Set 1: 0,00143 ms
- Set 2: 0,00105 ms
- Set 3: 0,00111 ms
- Set 4: 0,00102 ms

Como los valores obtenidos son extremadamente pequeños, a fines prácticos se realiza una conversión de milisegundos (ms) a microsegundos (μs) para que las diferencias puedan apreciarse con mayor claridad en el gráfico y evitar valores cercanos a cero.

- Set 1: 1,43 μs
- Set 2: 1,05 μs
- Set 3: 1,11 μs
- Set 4: 1,02 μs

Estos resultados convertidos son reflejados en el siguiente gráfico:

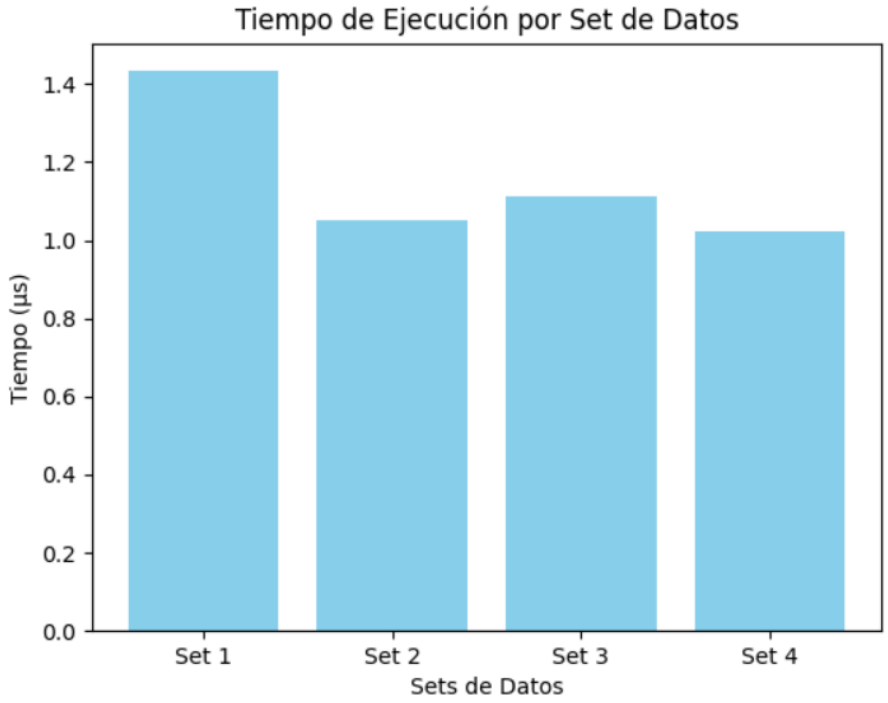

Figura 5: Comparación de los tiempos de ejecución del algoritmo para 4 sets de datos.

4.8. Informe de Resultados

En el gráfico se puede ver el comportamiento de los tiempos de ejecución de los sets de datos registrados. Se puede observar que todos los valores se mantienen en un rango reducido (entre 1

y 1,50 ms), esto concuerda con la complejidad temporal lineal analizada anteriormente. Como los sets de datos tienen tamaños comparables, los tiempos no muestran diferencias significativas para este algoritmo dinámico.

Para visualizar la comparación entre los tiempos medidos y la complejidad del algoritmo, se superpuso una curva de referencia k * n ajustada a los datos. Se observa que los tiempos siguen una tendencia lineal, aunque en n=6 se registra una caída respecto de la recta teórica. Sin embargo, dado que la diferencia es del orden de microsegundos, no representa una desviación significativa en términos prácticos. En general, la comparación es consistente con un crecimiento lineal, tal como predice la complejidad temporal del algoritmo dinámico

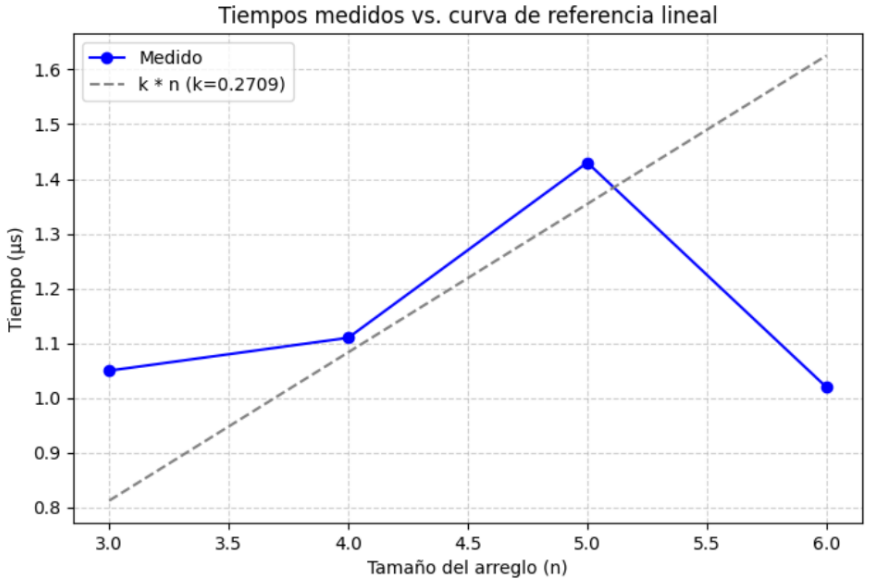

Figura 6: Comparación entre los tiempos medidos y la curva de referencia lineal

Teóricamente, el algoritmo de Backtracking presenta una complejidad temporal de $\mathbf{O(n^2)}$ en el peor caso, ya que debe evaluar todas las posibles subsecuencias contiguas del arreglo, realizando operaciones constantes por cada una de ellas. En cambio, el algoritmo usando programación dinámica alcanza una complejidad $\mathbf{O(n)}$, al aprovechar resultados previamente calculados y recorrer el arreglo una única vez. Por lo tanto, el enfoque dinámico resulta significativamente más eficiente y escalable.
Para los tiempos de ejecución, los tiempos medidos reflejan esta diferencia: el algoritmo de backtracking se registró valores entre **2,88 ms y 10,33 ms**, mientras que el dinámico se obtuvo tiempos entre **1 y 1,4** μ**s**. Esto implica una mejora de varios órdenes de magnitud en favor del enfoque dinámico.
Esto puede ser porque la diferencia radica en la cantidad de operaciones realizadas: el algoritmo dinámico mantiene un número constante de operaciones por elemento, mientras que el de backtracking explora numerosas combinaciones posibles, incluso con podas, lo que incrementa considerablemente el tiempo total de ejecución.

5. Anexo

5.1. Referencias

- GeeksforGeeks. (2025, July 23). Number of subarrays having even product. https://www.geeksforgeeks.org/dsa/number-of-subarrays-having-even-product/
- AfterAcademy. (2019, October 16). Maximum subarray sum – interview problem. https://afteracademy.com/blog/maximum-subarray-sum/