Integrantes:
Juan Manuel Ramirez
Juan Sebastian Poveda
Maria Paz Henao

Complejidad Temporal

Línea	Código	Costo
1	`HashMap<Integer, Integer> sumMap = new HashMap<>();`	O(1)
2	`int currentSum = 0;`	O(1)
3	`for (int i = 0; i < arr.length; i++) {`	O(n)
4	`currentSum += arr[i];`	O(1)
5	`if (currentSum == S) {`	O(1)
6	`return new int[]{0, i};}`	O(1)
7	`Integer prevIndex = sumMap.get(currentSum - S);`	O(1) ó O(n)
8	`if (prevIndex != null) {`	O(1)
9	`int startIndex = prevIndex + 1;`	O(1)
10	`return new int[]{startIndex, i};`	O(1)
11	`sumMap.put(currentSum, i);`	O(1) ó O(n)
12	`return new int[]{-1, -1};`	O(1)

Explicación costos

c1: inicialización de la tabla hash, se realiza una sola vez al principio del método
c2: inicialización de acumulador, se realiza una vez
c3: recorrido del arreglo, se repite n veces, pues es i = n ya no cumple la condición y no ingresa al bloque de código dentro del bucle.
c4: actualización del acumulador, es una asignación, por lo que su complejidad no depende de la cantidad de datos, se repite n-1 veces.
c5: evaluación si la suma acumulada es el objetivo, este caso es especial, es para evaluar si la suma acumulada se encuentra desde el índice 0 hasta i. Se repite n-1 veces.

c6: solo se ejecuta una vez si se cumple la condición anterior.
c7: obtención de llave (acumulador-objetivo), en el mejor caso para el hashmap es O(1), en su peor caso es O(n).Se repite n-1 veces.
c8: evaluación si se encontró una llave (acumulador-objetivo), se repite n-1 veces.
c9: asignación índice donde se cumple el criterio del problema, se ejecuta solo una vez cuando la condición se cumple.
c10: retorno de los índices que cumplen con la suma S. Se ejecuta solo una vez cuando se cumple la condición.
c11: pone la llave como el acumulador y el valor como índice, si hay colisiones en el peor caso la complejidad puede ser O(n), en el mejor caso donde no las hay es O(1).
c12: retorna índices fuera del alcance del arreglo indicando que no encontró el subarreglo que cumpla la suma S. Se repite una sola vez.

Mejor caso:
Al sumar las complejidades obtenemos : 11*O(1) + O(n) donde n es el tamaño del arreglo ingresado. Debido a que las constantes se encuentran acotadas por la función dependiente de n podemos concluir que la complejidad en el mejor caso es de O(n). Este caso no contempla las colisiones de la tabla hash.

Peor caso:
Al sumar las complejidades tenemos que la complejidad es 9*O(1) + 3*O(n). Teniendo en cuenta que c*O(n) = O(n) para cualquier constante c obtenemos que la complejidad del algoritmo es O(n) en el peor caso.

Complejidad espacial

(sub arreglo con suma objetivo)

Línea	Código	Costo
1	`HashMap<Integer, Integer> sumMap = new HashMap<>();`	O(n)
2	`int currentSum = 0;`	O(1)
3	`for (int i = 0; i < arr.length; i++) {`	O(1)
4	`currentSum += arr[i];`	O(1)
5	`if (currentSum == S) {`	O(1)
6	`return new int[]{0, i};}`	O(1)

7	`Integer prevIndex = sumMap.get(currentSum - S);`	O(1)
8	`if (prevIndex != null) {`	O(1)
9	`int startIndex = prevIndex + 1;`	O(1)
10	`return new int[]{startIndex, i};`	O(1)
11	`sumMap.put(currentSum, i);`	O(n)
12	`return new int[]{-1, -1};`	O(1)

Explicación de costos

c1: Se crea una nueva hash table capaz de almacenar n entradas por cada índice del arreglo.
c2: Se inicializa una variable escalar currentSum, que almacena un Integer que ocupa el mismo espacio y no depende de n.
c3: Se declara un ciclo for con una variable i, que es un simple contador que almacena el número de iteraciones del ciclo.
c4: Se actualiza la variable ya existente currentSum
c5: Se declara una condición if, que no crea ninguna variable ni ocupa espacio significativo
c6: Aunque se crea un nuevo arreglo, este cuenta con sólo 2 elementos; así que es un arreglo fijo.
c7: En esta línea se realiza una búsqueda en el mapa, solo se crea la variable prevIndex que es un escalar.
c8: Se declara una condición if, que evalúa que prevIndex no sea nulo. No crea ninguna variable ni ocupa espacio significativo.
c9: En esta línea se crea una variable startIndex, que solamente almacena un escalar resultado de una suma.
c10: Aunque en esta línea se cree un arreglo, este no depende de n, ya que solo almacena 2 elementos.
c11: La inserción de la entrada con clave CurrentSum y valor i es de complejidad espacial O(n), porque, aunque el tamaño de la hashMap en el espacio ya fue contado en la primera línea, si se introducen más entradas que el tamaño del hashMap, se tendría que hacer una redimensión del hashMap. Esta redimensión aumentaría el tamaño del hashMap en n elementos.
c12: Se crea un nuevo arreglo que almacena 2 elementos -1. Es un arreglo fijo, cuya complejidad espacial no depende de n.

Conclusión: En el algoritmo de sumar los elementos de un arreglo solo se crea una variable de complejidad espacial O(n), al momento de inicializar la tabla hash. Esto es debido a que, en el peor de los casos, la tabla hash almacenará una entrada por cada índice el arreglo **arr**.

Tipo	Variable	Tamaño de 1 valor atómico	Cantidad de valores atómicos	Complejidad espacial
Entrada	arr	Integer 32 bits (4 bytes)	n	O(n)
Entrada	s	Integer 32 bits (4 bytes)	1	O(1)
Auxiliar	currentSum	Integer 32 bits (4 bytes)	1	O(1)
Auxiliar	startIndex	Integer 32 bits (4 bytes)	1	O(1)
Auxiliar	i	Integer 32 bits (4 bytes)	1	O(1)
Auxiliar	prevIndex	Integer 32 bits (4 bytes)	1	O(1)
Auxiliar	sumMap	HashMap object → n×(Node(Integer,Integer)) + referencias internas del HashMap	n	O(n)
Salida	result	32 bits (4 bytes)	2	O(1)

Complejidad Espacial Total = Entrada+Auxiliar+Salida = n +(1 +1 + 1 + 1 +1 + n)+ 2 = n+5+n+2 = 2n +7 = O(n)
Complejidad Espacial Auxiliar = 1 = O(1)
Complejidad Espacial Auxiliar + Salida = 1 + 2 = 3 = O(n)