Trabajo Práctico Nro. 3

Contando Pares

Asignatura: Taller de Programación Java.

Docentes: Dra. Antonela Tommasel

Dr. Alejandro Corbellini

Dr. Juan Manuel Rodriguez

Integrantes (LU, apellido y nombre, e-mail):

- 248671, Magali Boulanger, maga.boulanger8@gmail.com
- 249479, Corino Joshua Yoel, joshuc98@gmail.com }

Boulanger, Magalí
Corino, Joshua Yoel

Introducción

Para este trabajo se pidió encontrar los pares de enteros en un arreglo cuya suma es igual a un número arbitrario (*target*). Debieron tomarse algunas consideraciones para la resolución, tales como: los valores del arreglo no respetan un rango, pueden ser valores arbitrarios, pueden ser negativos o positivos, el par debe contener elementos en distintas posiciones del arreglo, pueden encontrarse duplicados pero no deben contarse los pares invertidos.

Por ejemplo en el siguiente arreglo arr[] = {-1, 1, 5, 5, 7} y para un target igual a 6 se encontrará la siguiente solución: [(-1, 7), (1, 5), (1, 5)], pero no deben considerarse soluciones como: [(-1, 7), (1, 5), (1, 5), (5, 1), (5, 1), (7, -1)].

Desarrollo

En el diseño de la mejor solución posible pasamos por varias versiones:

Naïve

Consiste en recorrer cada elemento del arreglo y chequear si hay otro número que sumado de target. Es la solución más simple y por lo tanto menos eficiente. La tomaremos de base para realizar comparaciones de las siguientes alternativas que plantearemos.

```
public List<Pair> isSumIn(int[] data, int sum) {

        List<Pair> pairs = new ArrayList<>();

        for (int i = 0; i < data.length; i++)
            for (int j = i + 1; j < data.length; j++)
                if ((data[i] + data[j]) == sum)
                    pairs.add(new Pair(data[i],data[j]));

        return pairs;
}
```

Complejidad Temporal: $O(n^2)$
Espacio Auxiliar : O(1)

Boulanger, Magalí
Corino, Joshua Yoel

Naïve 2

Al igual que Naive, consiste en recorrer cada elemento del arreglo y chequear si existe otro número que al sumarlo de target, se diferencia en la implementación porque se utiliza un Stream para recorrer el arreglo de datos, y el filter para aplicar las condiciones.

```
public List<Pair> isSumIn(int[] data, int sum) {

        List<Pair> pairs = new ArrayList<>();

        IntStream.range(0,  data.length)
         .forEach(i -> IntStream.range(i+1,  data.length)
             .filter(j -> i != j && data[i] + data[j] == sum)
             .forEach(j -> pairs.add(new Pair(data[i], data[j])))
                 );

        return pairs;
}
```

Complejidad Temporal: $O(n^2)$
Espacio Auxiliar : O(1)

SortSearch

Esta solución se basa en ordenar el arreglo y por cada valor data[i], buscar si target-data[i] se encuentra en el arreglo. Es necesario el ordenamiento porque se aplica búsqueda binaria. Esta solución resulta una buena opción y mejora bastante la complejidad respecto a la solución anterior dado que reduce el espacio de búsqueda, la búsqueda binaria tendrá una complejidad de O(log n).

Esta solución no resulta válida dado que presenta un problema, se trata de que no realiza un tratamiento correcto de los repetidos que no son considerados. Para compensar esta falla debe realizarse una modificación chequeando los valores vecinos y, en caso de que sean iguales, incluir la cantidad de pares repetidos en la solución.

Boulanger, Magalí
Corino, Joshua Yoel

```
    public List<Pair> isSumIn(int[] data, int sum) {

        List<Pair> pairs = new ArrayList<>();

        Arrays.sort(data);
        for (int i=0;i<data.length;i++){
            if(Arrays.binarySearch(data,sum-data[i])>0){
                pairs.add(new Pair(data[i],sum-data[i]));
            }
        }

        return pairs;
    }
```

Complejidad Temporal: O(n.log n)

Espacio Auxiliar : O(n)

Usando Map

1. Crear un mapa que almacene la frecuencia de cada número en el arreglo.
2. Para cada elemento chequear si puede ser combinado con otro elemento que no sea sí mismo.
3. Si encontramos el elemento, agregar los pares correspondientes.

Versión 1

Version 1.1

Esta fue la primera versión del algoritmo usando map, se implementó tener una aproximación a la mejora en el tiempo de ejecucion total que el map aportaría, no podría considerarse una solución al problema dado que fue implementado sin tomar en cuenta las condiciones de agregar un par en ambos sentidos o la de sumar un elemento consigo mismo. Se tomó de base el ejemplo anterior y por esto se mantiene la búsqueda binaria.

```
public List<Pair> isSumIn(int[] data, int sum) {

      List<Pair> pairs = new ArrayList<>();

      HashMap<Integer, Integer> frecuencyMap = new HashMap<>();
      int n=data.length;
```

```
	Arrays.sort(data);

	for (int i=0; i<n; i++){
		if(!frecuencyMap.containsKey(data[i]))
			frecuencyMap.put(data[i],0);
		frecuencyMap.put(data[i], frecuencyMap.get(data[i])+1);
	}

	for (int i=0;i<data.length;i++){
		if(Arrays.binarySearch(data,sum-data[i])>0){
				for (int k = 0; k < frecuencyMap.get(sum -
data[i]); k++) {
					pairs.add(new Pair(data[i], sum -
data[i]));
				}
		}
	}
	return pairs;
}
```

Complejidad Temporal: $O(n^2)$

Espacio Auxiliar : O(n)

Version 1.2

En esta sacamos del algoritmo el ordenamiento y la búsqueda binaria, para reemplazar todo por operaciones de mapa, obteniendo así una mejora del `frecuencyMap.get(sum-data[i]) != null` frente a `Arrays.binarySearch(data,sum-data[i])>0` ya que el get en un HashMap tiene complejidad temporal O(1) y la búsqueda binaria O(n.log n).

```
public List<Pair> isSumIn(int[] data, int sum) {

	List<Pair> pairs = new ArrayList<>();
	HashMap<Integer, Integer> frecuencyMap = new HashMap<>();
	int n=data.length;

	for (int i=0; i<n; i++){
		if(!frecuencyMap.containsKey(data[i]))
			frecuencyMap.put(data[i],0);
		frecuencyMap.put(data[i], frecuencyMap.get(data[i])+1);
	}
```

```
      for (int i=0; i<n; i++)
      {
            if(frecuencyMap.get(sum-data[i]) != null) {
                  for (int k = 0; k < frecuencyMap.get(sum - data[i]);
k++) {
                        pairs.add(new Pair(data[i], sum - data[i]));
                  }

            }
      }
      return pairs;
}
```

Complejidad Temporal: $O(n^2)$

Espacio Auxiliar : O(n)

Version 1.3

Se agregaron las condiciones necesarias para la correcta resolución del problema.

```
    public List<Pair> isSumIn(int[] data, int sum) {

        List<Pair> pairs = new ArrayList<>();
        HashMap<Integer, Integer> frecuencyMap = new HashMap<>();
        int n=data.length;

        for (int i=0; i<n; i++){
            if(!frecuencyMap.containsKey(data[i]))
                frecuencyMap.put(data[i],0);
            frecuencyMap.put(data[i], frecuencyMap.get(data[i])+1);
        }

        for (int i=0; i<n; i++)
        {
            if(frecuencyMap.get(sum-data[i]) != null) {
                if(!pairs.contains(new Pair(sum - data[i],data[i]))){
                    for (int k = 0; k < frecuencyMap.get(sum - data[i]);
k++) {
                        pairs.add(new Pair(data[i], sum - data[i]));
                    }
```

```
                    if (sum - data[i] == data[i])
                        pairs.remove(new Pair(data[i], sum - data[i]));
                }
            }
        }
        return pairs;
    }
```

Complejidad Temporal: $O(n^2)$

Espacio Auxiliar : O(n)

Versión 2

Otra posible implementación que se tuvo en cuenta fue la que se puede ver a continuación, la cual surgió modificando la versión anterior utilizando map. En este caso se almacenaron los pares encontrados en otro map extra, seguido de la cantidad de repeticiones del mismo. Luego se agregaron dichos pares a la lista retornada por el método. Con el objetivo de evitar recorrer toda la lista pairs para chequear que no contenga agregado el par invertido y hacer solo un get en el hash con complejidad O(1), mejorando la complejidad O(n) de recorrer una lista.

```
    public List<Pair> isSumIn(int[] data, int sum) {

        List<Pair> pairs = new ArrayList<>();
        HashMap<Integer, Integer> frecuencyMap = new HashMap<>();
        HashMap<Pair, Integer> pairMap = new HashMap<>();

        int n=data.length;

        for (int i=0; i<n; i++){
            if(!frecuencyMap.containsKey(data[i]))
                frecuencyMap.put(data[i],0);
            frecuencyMap.put(data[i], frecuencyMap.get(data[i])+1);
        }

        for (int i=0; i<n; i++)
        {
            Pair thisPair=new Pair(data[i],sum - data[i]);
            if(frecuencyMap.get(sum-data[i]) != null) {

                if(pairMap.get(new Pair(sum - data[i],data[i])) ==
null){
                    if(pairMap.get(thisPair)==null){
                        pairMap.put(thisPair,0);
```

```
                    }

pairMap.put(thisPair,pairMap.get(thisPair)+frecuencyMap.get(sum -
data[i]));
                    if (sum - data[i] == data[i])
                        pairMap.put(thisPair,pairMap.get(thisPair)-1);
                }
            }
        }

       for (Pair p:pairMap.keySet()) {
            for(int j=0;j<pairMap.get(p);j++){
                pairs.add(new Pair(p.getI(),p.getJ()));
            }
        }
        return pairs;

    }
```

Complejidad Temporal: $O(n^2)$

Espacio Auxiliar : O(n)

Comparación de consumo de memoria y tiempo

Benchmark

Se utilizó el framework de microbenchmarking JMH, el cual nos permite construir, ejecutar y analizar nano/micro/mili/macro benchmarks escritos en Java y otros lenguajes de la JVM. El benchmarking es una técnica para medir el rendimiento de un sistema o componente del mismo, en este caso la utilizaremos para comparar el tiempo que de los algoritmos implementados tardan en resolver el problema.
JMH permite correr los benchmarks es diferentes modos, este modo le indica a JMH qué se quiere medir. Los modos posibles son los siguientes:

- Throughput: Mide el número de operaciones por segundo, es decir, el número de veces por segundo que el benchmark puede ser ejecutado.
- Average Time: Mide el promedio de repeticiones que le toma al benchmark ejecutar (una vez).
- Sample Time: Mide cuánto tiempo le toma al benchmark ejecutarse incluyendo tiempos máximos, tiempos mínimos, etc..
- Single Shot Time: Mide cuánto tiempo le toma al benchmark la ejecución.
- All: Mide todo.

El modo por defecto es Throughput.

En este trabajo seleccionamos el modo Average Time para nuestro benchmark. Y se corrió por defecto, ejecutando 5 veces con 10 iteraciones cada uno, considerando 5 de warm up y 5 de medición.

Para las pruebas se creó un proyecto nuevo jmh-java-benchmark desde maven y se importó el proyecto con las implementaciones de las diferentes soluciones.Y se corrieron en el siguiente entorno:

- Hardware: PC de escritorio con Intel Core i5-7400 a 3.00 GHz, 16 GB de RAM a 2333 Mhz DDR4
- Sistema operativo: Windows 10
- Entorno de desarrollo: IntelliJ Ultimate 2020.1
- JMH 1.23
- Java 8
- Apache Maven 3.8.0
- JDK 1.8.0_171

Resultados Obtenidos

Para introducir el análisis de resultados se debe aclarar que se hicieron pruebas con arreglos de distintos tamaños y el elegido para la comparación fue de size=100000, ya que es la magnitud donde todos los algoritmos corren un tiempo aceptable, si incrementamos a 1000000, el algoritmo NaiveTwo que es el más lento y que escala temporalmente n^2 pasa a tardar demasiado y se vuelve tedioso esperar las ejecuciones.

Memoria

- Naive: 30.5 MB
- NaiveTwo: 85 MB
- SortSearch: 1300 MB
- Map: 900 MB
- MapTwo: 1065 MB

Tiempos

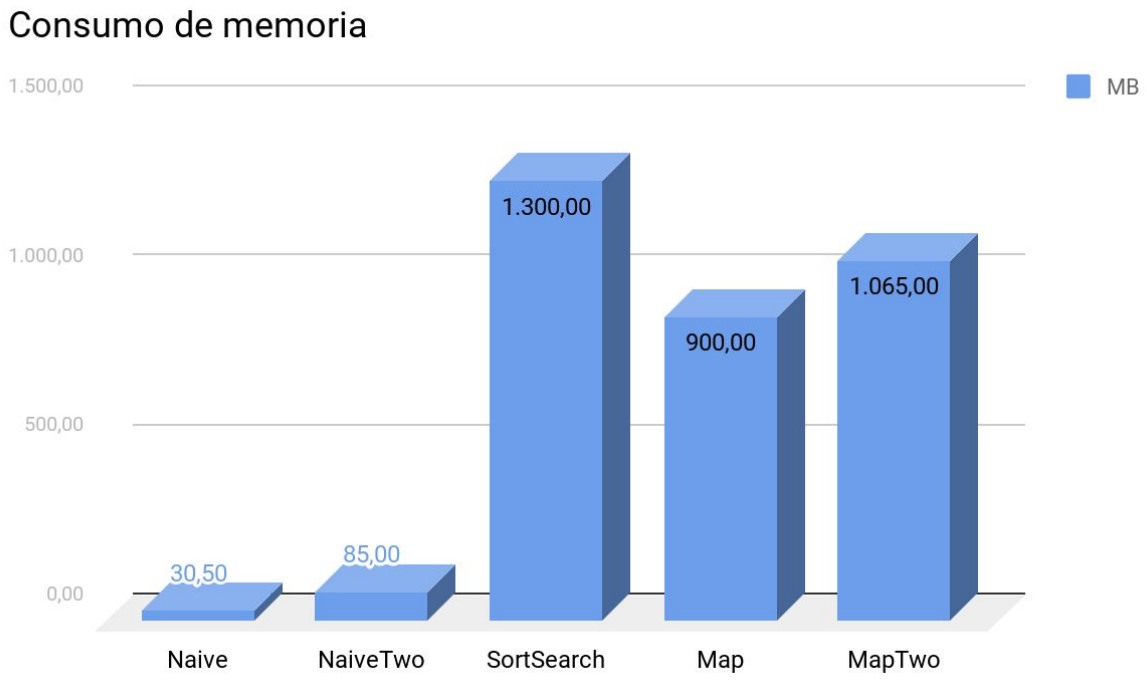
- Naive:
    - 3,019 ±(99.9%) 0,065 s/op [Average]
    - Total time: 00:09:59

- NaiveTwo:
  - 15,038 ±(99.9%) 1,931 s/op [Average]
  - Total time: 00:13:04
- SortSearch:
  - 0,013 ±(99.9%) 0,001 s/op [Average]
  - Total time: 00:08:28
- Map:
  - 0,018 ±(99.9%) 0,001 s/op [Average]
  - Total time: 00:08:21
- MapTwo:
  - 0,017 ±(99.9%) 0,001 s/op [Average]
  - Total time: 00:08:17

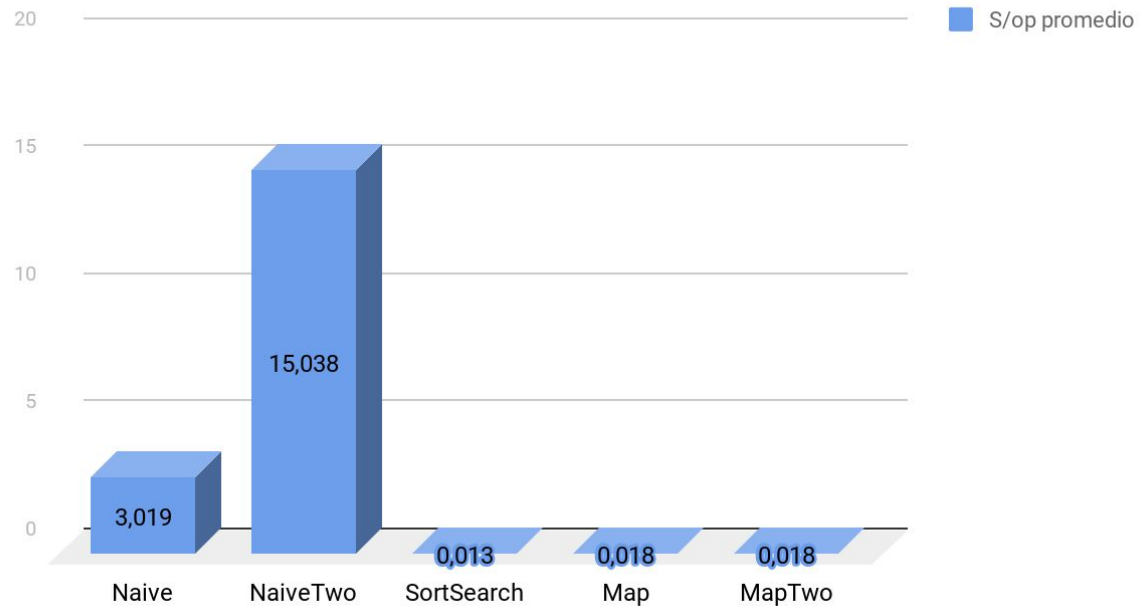
Tiempos con size 1000000

- Map:
  - 0,601 ±(99.9%) 0,018 s/op [Average]
  - Total time: 00:08:40
- MapTwo:
  - 0,622 ±(99.9%) 0,020 s/op [Average]
  - Total time: 00:08:40

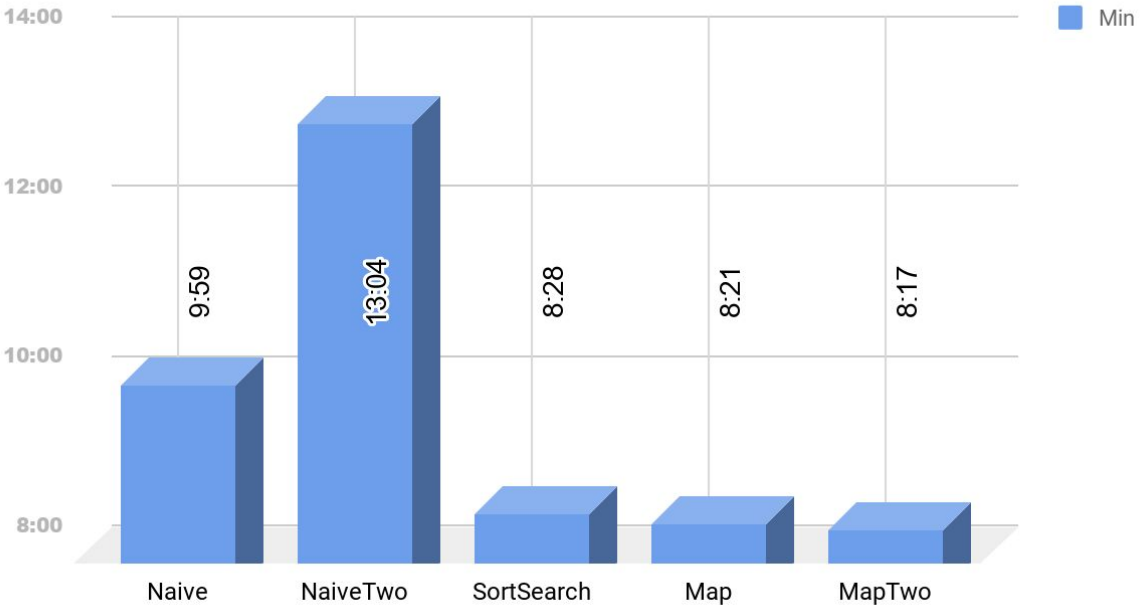
Gráficos



Tiempo promedio



Tiempo total



Boulanger, Magalí
Corino, Joshua Yoel

Conclusión

Analizando los valores obtenidos podemos concluir que las mejores alternativas son Map y MapTwo. El motivo que nos conduce a esta deducción es que sus tiempos totales y sus cantidades de operaciones por segundo, mejoran notablemente respecto a los resultados de Naive y NaiveTwo, no es así respecto a SortSearch que obtiene valores temporales similares, pero supera a todos en consumo de memoria y por esta razón, sumado a que la implementación no resuelve correctamente el problema ya que como dijimos, no considera repetidos; esto podría haberse considerado e implementado, pero no la llevamos a cabo porque aun así, su tiempo no mejoraría y la memoria ocupada tampoco disminuiría.

Comparando las dos mejores alternativas podemos ver que si bien temporalmente con size=100000 no presentan grandes diferencias, respecto a la memoria ocupada si se presenta una divergencia, donde MapTwo ocupa un 18,3% más de memoria que Map debido a que usa temporalmente un mapa para ir almacenando los pares y la otra implementación guarda los pares directamente en el ArrayList a retornar. Para intentar comparar sus tiempos con más precisión se corrieron los algoritmos con un problema de tamaño 1000000, y los resultados fueron similares pero el Map, superó al MapTwo también en tiempo por una mínima diferencia, quedando este como el mejor de los algoritmos implementados.