

# 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 }

---

# Índice

<b>Índice</b>	<b>1</b>
<b>Introducción</b>	<b>3</b>
<b>Desarrollo</b>	<b>3</b>
Naïve	3
Naïve 2	4
SortSearch	4
Usando Map	6
Versión 1	6
Versión 1.1	6
Versión 1.2	7
Versión 1.3	8
Versión 1 Fixed	9
Versión 2	10
Versión 2 Fixed	11
Versión 3	12
<b>Comparación de consumo de memoria y tiempo</b>	<b>13</b>
Benchmark	13
Resultados Obtenidos	14
Memoria	15
Size 100.000:	15
Size 500.000:	15
Size 1.000.000:	15
Size 5.000.000:	15
Mediciones de memoria representativas	16
Tiempos	17
Tiempos con size 100.000	17
Tiempos con size 500.000	18
Tiempos con size 1.000.000	19
Tiempos con size 5.000.000	19
Gráficos	20
Consumo de memoria para size 100.000 según JVM	20
Consumo de memoria para size 100.000 según Runtime	20
Tiempo promedio con size 100.000	21
Tiempo total para size 100.000	21
Consumo de memoria para size 500.000 según Runtime	22
Tiempo promedio con size 500.000	22
Consumo de memoria para size 1.000.000 según JVM	23

---

Consumo de memoria según Runtime para size 1.000.000	23
Tiempo promedio con size 1.000.000	24
Tiempo total para size 1.000.000	24
<b>Conclusiones</b>	<b>26</b>
Conclusión anterior	26
Conclusión final	26

## 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:

Dado que se trata de dos estructuras *for* anidadas, para expresar la cota en Big Oh, nos fijamos en el peor de los casos: se ciclará  $n$  veces (correspondientes al *for* exterior) por  $n$  veces (correspondientes al *for* interior), donde  $n$  se corresponde con la cantidad de elementos en el vector *data*. Por lo tanto la complejidad final es de  $O(n^2)$ .

Espacio Auxiliar :  $O(1)$

## 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:

En este caso, la complejidad temporal es igual que la de Naïve, esto se debe a que en esta versión, cambia la forma en que se recorre *data* pero no los límites con los que se realiza dicho recorrido, por lo tanto continúa teniendo una complejidad de  $O(n^2)$ .

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.

```
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:

Para este algoritmo, la complejidad es  $O(n \cdot \log n)$ , ya que al ordenar el arreglo primero nos ahorramos el segundo for reemplazandolo por una búsqueda binaria, así, nos queda el for  $n$  por la complejidad de la búsqueda binaria que es  $\log n$ .

Para calcular la complejidad de la búsqueda binaria: sabemos que inicialmente en número de elementos por analizar es  $n$ , tras la primera división, el número será como mucho  $n/2$  (porque nos quedamos con la mitad de elementos); tras la segunda división, el número será como mucho  $n/4$ ; y así sucesivamente. Entonces, tras la división número  $i$ , el número de elementos por analizar será como mucho:  $n / 2^i$ . Sabemos que el peor caso se da cuando el elemento a buscar no se encuentra en el vector (cuando tras dividir los elementos por analizar nos quedemos con un número menor a 1):  $n / 2^m < 1$ . Transformando esta fórmula a un logaritmo en base 2, tenemos que:  $n < 2^m$ , por ende  $\log n < m$ .

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

#### Versión 1.1

Esta fue la primera versión del algoritmo usando map, se implementó tener una aproximación a la mejora en el tiempo de ejecución 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> frequencyMap = new HashMap<>();  
    int n=data.length;  
    Arrays.sort(data);  
  
    for (int i=0; i<n; i++){  
        if(!frequencyMap.containsKey(data[i]))  
            frequencyMap.put(data[i],0);  
        frequencyMap.put(data[i], frequencyMap.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 < frequencyMap.get(sum -  
data[i]); k++) {  
                pairs.add(new Pair(data[i], sum -  
data[i]));  
            }  
        }  
    }  
    return pairs;  
}
```

### Complejidad Temporal:

En este caso vuelven a encontrarse dos *for* anidados, el primero cicla  $n$  veces, donde  $n$  son cada uno de los datos en *data*, y el interno lo hace según la cantidad de apariciones del número que sumado al dato tomado en el primer *for* da como resultado la suma buscada. En el peor de los casos, el *for* interno también ciclará un número de veces igual a la cantidad de datos en *data*, esto ocurrirá cuando *data* sea un vector con todos sus elementos iguales y la suma buscada sea el doble de dicho elemento. Tomando este caso como referencia, la complejidad será  $O(n^2)$ .

Espacio Auxiliar :  $O(n)$ .

### Versión 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 *frequencyMap.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  $O(n \cdot \log n)$ .

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

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

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

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



Complejidad Temporal:

Si bien se realizan cambios, la fracción de código que define la complejidad del algoritmo no varía en sus límites, es por esto que la complejidad en el peor de los casos será  $O(n^2)$ , por la misma causa mencionada en la versión anterior.

Espacio Auxiliar :  $O(n)$

### Versión 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> frequencyMap = new HashMap<>();
    int n=data.length;

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

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

Las modificaciones realizadas no provocan cambios sobre la complejidad final del algoritmo, es por esto que sigue siendo  $O(n^2)$ .

Espacio Auxiliar :  $O(n)$ .

### Versión 1 Fixed

Para reducir la complejidad del método *contains* de *List* se decidió tomar un orden sobre los pares encontrados, donde se almacenan nuevos pares solo si el primer elemento es mayor al segundo, así, se asegura que si el elemento que buscamos *sum - data[i]* es mayor a *data[i]* entonces el par ya fue agregado o se agregara en futuras iteraciones. Por otra parte, como consecuencia de esta decisión, quedarían excluidos aquellos pares formados por elementos correspondientes al mismo valor, en este caso si se utilizará la función *contains* para chequear si dichos pares ya han sido agregados a la lista.

```
public List<Pair> isSumIn(int[] data, int sum) {
    List<Pair> pairs = new ArrayList<>();
    HashMap<Integer, Integer> frequencyMap = new HashMap<>();
    int n=data.length;
    // Save elements in map
    for (int i=0; i<n; i++){
        if(!frequencyMap.containsKey(data[i]))
            frequencyMap.put(data[i],0);
        frequencyMap.put(data[i], frequencyMap.get(data[i])+1);
    }
    // Iterate through each element and add a pair if (sum-data[i]) !=
    null (every pair is counted twice)
    for (int i=0; i<n; i++)
    {
        if(frequencyMap.get(sum-data[i]) != null) {
            if ((sum - data[i] < data[i]) || ((sum - data[i] == data[i])
            && (!pairs.contains(new Pair(sum - data[i],data[i]))))) {
                for (int k = 0; k < frequencyMap.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;
}
```

## 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> frequencyMap = new HashMap<>();
    HashMap<Pair, Integer> pairMap = new HashMap<>();

    int n=data.length;

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

    for (int i=0; i<n; i++)
    {
        Pair thisPair=new Pair(data[i],sum - data[i]);
        if(frequencyMap.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)+frequencyMap.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:

En el peor de los casos, las últimas estructuras *for* anidadas que generan la lista de retorno, son aquellas que determinan la complejidad, la cual es  $O(n^2)$ , dado que si ocurriese el caso descrito en la versión 1.1 se recorrerían igualmente todos los datos.

Espacio Auxiliar :  $O(n)$

## Versión 2 Fixed

En esta nueva versión se tomó en cuenta uno de los comentarios de la cátedra luego de la entrega del trabajo, donde se mencionaba que se creaban muchos objetos innecesariamente. Para evitar esto, se creó el objeto *thisPair* sólo si este era necesario, y por otra parte, se añadieron a la lista que se retorna referencias a los pares ya almacenados en el hash, para evitar la creación de nuevos objetos sin sentido.

```
public List<Pair> isSumIn(int[] data, int sum) {
    List<Pair> pairs = new ArrayList<>();
    HashMap<Integer, Integer> frequencyMap = new HashMap<>();
    HashMap<Pair, Integer> pairMap = new HashMap<>();
    int n=data.length;
    for (int i=0; i<n; i++){
        if(!frequencyMap.containsKey(data[i]))
            frequencyMap.put(data[i],0);
        frequencyMap.put(data[i], frequencyMap.get(data[i])+1);
    }
    for (int i=0; i<n; i++)
    {
        if(frequencyMap.get(sum-data[i]) != null) {
            if(pairMap.get(new Pair(sum - data[i],data[i])) == null){
                Pair thisPair=new Pair(data[i],sum - data[i]);
                pairMap.putIfAbsent(thisPair, 0);
                pairMap.put(thisPair,pairMap.get(thisPair)+
frequencyMap.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(p);  
    }  
  }  
  return pairs;  
}
```

Complejidad Temporal:

Los cambios no modifican la complejidad temporal, por lo tanto se mantiene igual que la versión anterior,  $O(n^2)$ .

Espacio Auxiliar:  $O(n)$ .

### Versión 3

En esta nueva versión se introduce un cambio en las estructuras utilizadas, reemplazando ArrayList por LinkedList, dado que esta última presenta un mejor desempeño a la hora de agregar elementos.

Otra modificación introducida fue el uso de Trove, una librería con todas las funcionalidades de Java Collections que permite el uso de primitivos, haciendo nuestras colecciones más livianas y rápidas. Así, cambiamos los mapas por mapas de trove, esto introduce una mejora de rendimiento significativa dado que requiere mucho menos espacio:

```
public List<Pair> isSumIn(int[] data, int sum) {  
    List<Pair> pairs = new LinkedList<>();  
    TIntIntHashMap frequencyMap= new TIntIntHashMap();  
    TObjectIntHashMap pairMap = new TObjectIntHashMap();  
    int n=data.length;  
    for (int i=0; i<n; i++){  
        if(!frequencyMap.containsKey(data[i]))  
            frequencyMap.put(data[i],0);  
        frequencyMap.put(data[i], frequencyMap.get(data[i])+1);  
    }  
    for (int i=0; i<n; i++)  
    {  
        if(frequencyMap.get(sum-data[i]) !=  
frequencyMap.getNoEntryValue()) {
```

```
        if(pairMap.get(new Pair(sum - data[i],data[i])) ==  
pairMap.getNoEntryValue()){  
            Pair thisPair=new Pair(data[i],sum - data[i]);  
            pairMap.putIfAbsent(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 (Object p:pairMap.keySet()) {  
    for(int j=0;j<pairMap.get(p);j++){  
        pairs.add((Pair)p);  
    }  
}  
return pairs;  
}
```

Complejidad Temporal:

Se mantiene igual que la versión anterior,  $O(n^2)$ .

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 en 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 cada prueba se modificaron, dentro de *MyBenchmark.java*, el tamaño del problema y el algoritmo a utilizar para resolverlo. Dichas configuraciones se encuentran comentadas en el código fuente para poder probar cada una de las mismas.

Para las pruebas se creó un proyecto nuevo *jmh-java-benchmark* desde Maven y se importó el otro proyecto agregado previamente al repositorio local, que tiene las implementaciones de las diferentes soluciones. El benchmark está implementado en la clase *MyBenchmark.java* y consta de dos partes:

- La clase *MyState*, que está precedida por *@State*, que le indica a JMH que en esta clase se hará el setup. Aquí se declaran las variables necesarias y, en el constructor, se inicializan las mismas y se generan los datos de prueba.
- La otra parte es el método nombrado *testMethod* es el que finalmente se benchmarkea, donde se llama al problem solver declarado en el state para resolver el problema.

Los benches 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 inicial fue de  $\text{size}=100000$ , 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

Una de las mediciones la hicimos sobre la memoria ocupada por la JVM, ya que si bien la memoria que esta pide al sistema operativo puede no ser totalmente representativa, consideramos que como estos algoritmos se corren en Java, la JVM siempre estará presente, por lo tanto, aunque por efectos de boxing/unboxing o por retrasar el uso de GC pueda pedir en casos más memoria de la necesaria, es un parámetro al fin:

Size 100.000:

- Naive: 30.5 MB
- NaiveTwo: 34 MB
- SortSearch: 31 MB
- Map: 900 MB
- MapFixed: 1067 MB
- MapTwo: 1065 MB
- MapTwoFixed: 1065 MB
- MapThree: 120 MB

Size 500.000:

- Naive: 120 MB
- NaiveTwo: 120 MB
- SortSearch: 122 MB
- Map: 1300 MB
- MapFixed: 1471 MB
- MapTwo: 1470 MB
- MapTwoFixed: 1470 MB
- MapThree: 270 MB

Size 1.000.000:

- Map: 1541 MB
- MapFixed: 1383 MB
- MapTwo: 1473 MB
- MapTwoFixed: 1574 MB
- MapThree: 1340 MB

Size 5.000.000:

- Map: 1838 MB
- MapFixed: 2045 MB
- MapTwo: 1780 MB
- MapTwoFixed: 1800 MB



- MapThree: 1824 MB

## Mediciones de memoria representativas

En esta nueva entrega, ya que con la aclaración de la cátedra nos dimos cuenta que podríamos mejorar el análisis al obtener una medida más representativa del espacio de memoria requerido realmente, realizamos pruebas diferentes con una medición que se incluyó dentro de *solutions*. El código utilizado es el siguiente:

```
public static void main(String[] args) {

    ProblemGen problemGen = new ProblemGen();
    ArrayList<IPProblemSolver> solutors = new
ArrayList<IPProblemSolver>();
    solutors.add(new SolutionNaive());
    solutors.add(new SolutionNaive2());
    solutors.add(new SolutionMap());
    solutors.add(new SolutionMapFixed());
    solutors.add(new SolutionMapTwo());
    solutors.add(new SolutionMapTwoFixed());
    solutors.add(new SolutionMapThree());
    solutors.add(new SolutionSortSearch());
    problemGen.genRandomProblem(1000000);
    int random=(int)(Math.random() * 2 * Integer.MAX_VALUE +
Integer.MIN_VALUE/2);
    for (IPProblemSolver i: solutors){
        i.isSumIn(problemGen.getData(), random);
        Runtime.getRuntime().gc();
    }
}
```

Este fragmento de código dentro de *Solutions.java* crea una lista con todas las soluciones y luego, una a una se ejecuta el método *isSumIn* correspondiente a cada implementación sobre el mismo problema. Luego de la ejecución de cada algoritmo se realiza un llamado al Garbage Collector para liberar la memoria ocupada por dicha ejecución, si bien sabemos que el llamado mediante “.gc” es solo una sugerencia a Java y puede no realizarlo en el instante, comprobamos que así sea.

Por otro lado, dentro de cada una de las implementaciones de las soluciones, añadimos el siguiente código al principio y al final para ver con cuánta memoria se entra al algoritmo y cuanta más usa este así, sacando la diferencia, obtenemos puramente la memoria usada usando la clase Runtime de java que permite conectarnos al entorno donde la app se está corriendo:

```
Runtime runtime = Runtime.getRuntime();
    System.out.println("Memory used "+this.toString() + " "+
(runtime.totalMemory() - runtime.freeMemory())/ (1024*1024) + "MB");

...

System.out.println("Memory used "+this.toString() + " "+
(runtime.totalMemory() - runtime.freeMemory())/ (1024*1024) + "MB");
```

Se hicieron pruebas con problemas de tamaño 100000, 500000, 1000000 y 5000000, y los resultados obtenidos fueron los siguientes:

Algoritmo \ Size	100.000	500.000	1.000.000	5.000.000
Naive	1 MB	1 MB	2 MB	-----
Naive 2	26 MB	54 MB	40 MB	-----
SortSearch	0 MB	0 MB	0 MB	0 MB
Map	14 MB	38 MB	110 MB	460 MB
MapFixed	13 MB	37 MB	92 MB	459 MB
MapTwo	14 MB	73 MB	183 MB	529 MB
MapTwoFixed	13 MB	70 MB	157 MB	458 MB
MapThree	6 MB	28 MB	55 MB	224 MB

## Tiempos

### Tiempos con size 100.000

- 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:
  - $0,013 \pm(99.9\%) 0,001$  s/op [Average]
  - Total time: 00:08:28
- Map:
  - $0,018 \pm(99.9\%) 0,001$  s/op [Average]
  - Total time: 00:08:21
- MapFixed:
  - $0,016 \pm(99.9\%) 0,001$  s/op [Average]
  - Total time: 00:08:25
- MapTwo:
  - $0,017 \pm(99.9\%) 0,001$  s/op [Average]
  - Total time: 00:08:17
- MapTwoFixed:
  - $0,014 \pm(99.9\%) 0,001$  s/op [Average]
  - Total time: 00:08:25
- MapThree:
  - $0,010 \pm(99.9\%) 0,001$  s/op [Average]
  - Total time: 00:08:25

### **Tiempos con size 500.000**

- Map:
  - $0,256 \pm(99.9\%) 0,042$  s/op [Average]
- MapFixed:
  - $0,245 \pm(99.9\%) 0,009$  s/op [Average]
- MapTwo:
  - $0,241 \pm(99.9\%) 0,006$  s/op [Average]
- MapTwoFixed:
  - $0,237 \pm(99.9\%) 0,014$  s/op [Average]
- MapThree:

- 
- $0,065 \pm(99.9\%) 0,003$  s/op [Average]

### Tiempos con size 1.000.000

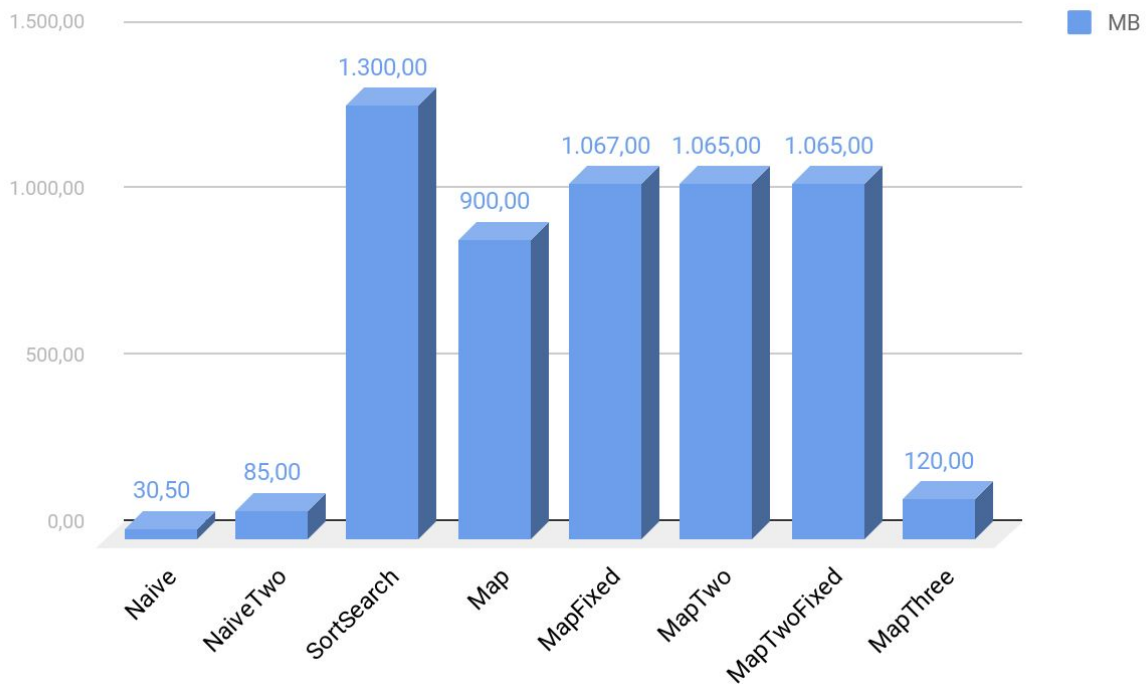
- Map:
  - $0,601 \pm(99.9\%) 0,018$  s/op [Average]
  - Total time: 00:08:40
- MapFixed:
  - $0,606 \pm(99.9\%) 0,033$  s/op [Average]
  - Total time: 00:08:41
- MapTwo:
  - $0,622 \pm(99.9\%) 0,020$  s/op [Average]
  - Total time: 00:08:40
- MapTwoFixed:
  - $0,560 \pm(99.9\%) 0,024$  s/op [Average]
  - Total time: 00:08:38
- MapThree:
  - $0,144 \pm(99.9\%) 0,005$  s/op [Average]
  - Total time: 00:08:27

### Tiempos con size 5.000.000

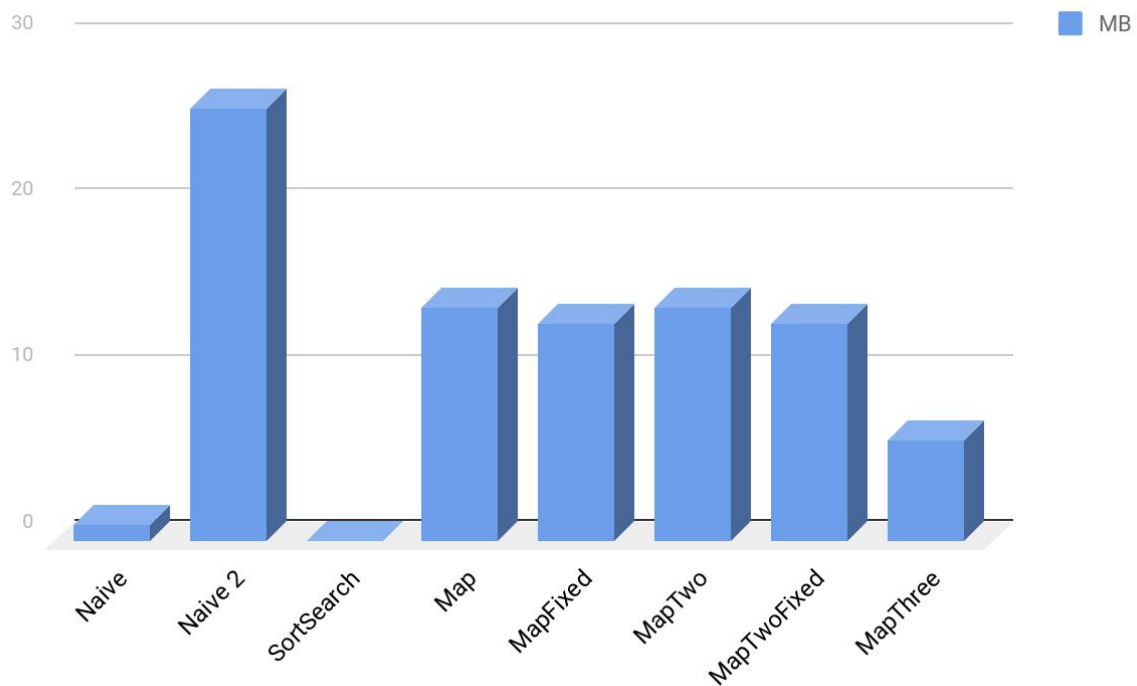
- Map:
  - $3,863 \pm(99.9\%) 0,599$  s/op [Average]
- MapFixed:
  - $4,035 \pm(99.9\%) 0,458$  s/op [Average]
- MapTwo:
  - $3,971 \pm(99.9\%) 0,845$  s/op [Average]
- MapTwoFixed:
  - $4,061 \pm(99.9\%) 0,362$  s/op [Average]
- MapThree:
  - $0,955 \pm(99.9\%) 0,014$  s/op [Average]

## Gráficos

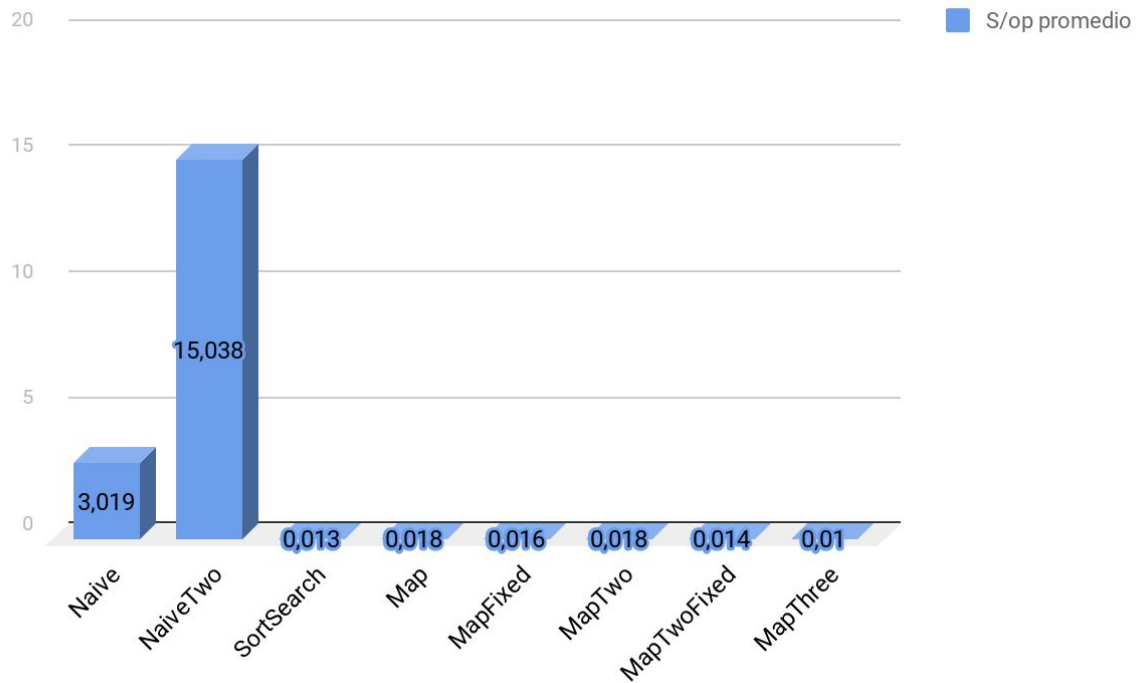
### Consumo de memoria para size 100.000 según JVM



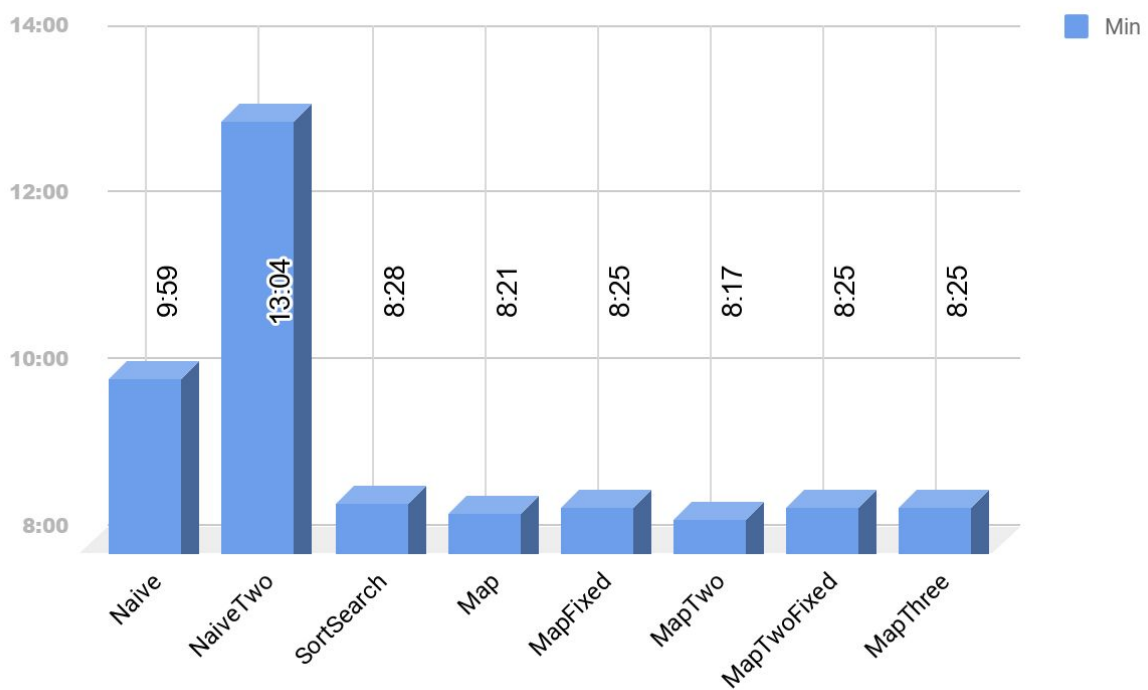
### Consumo de memoria para size 100.000 según Runtime



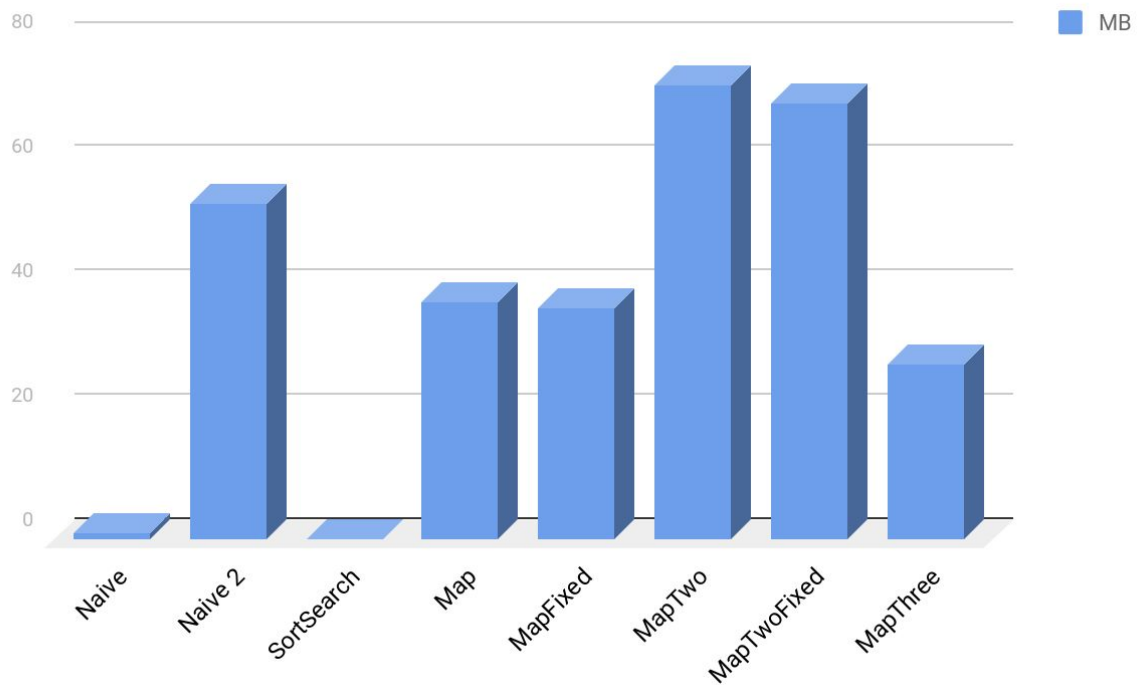
### Tiempo promedio con size 100.000



### Tiempo total para size 100.000

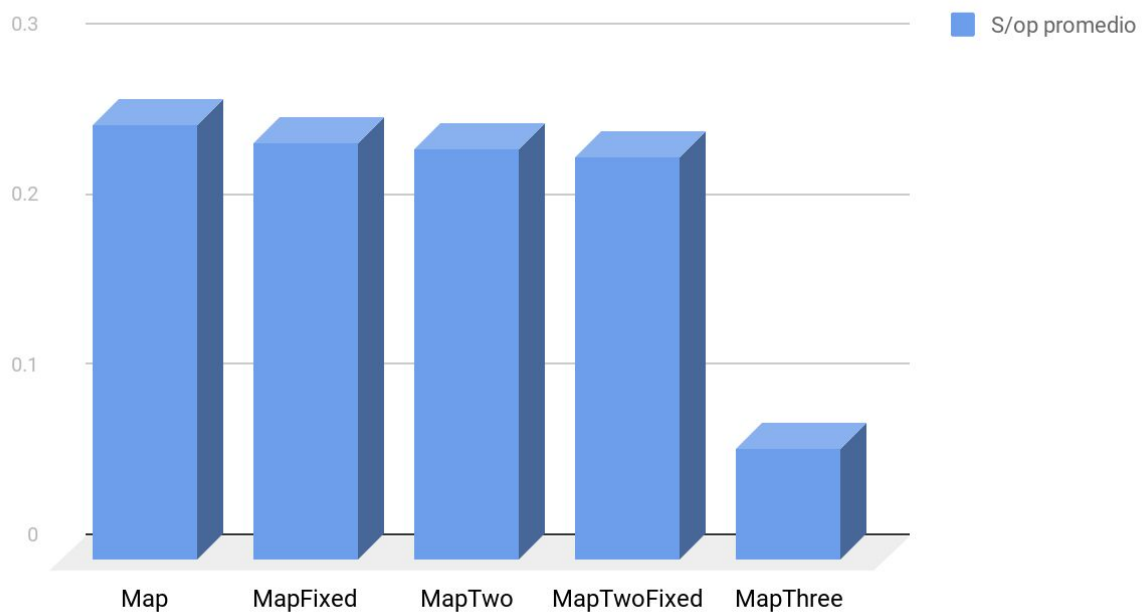


### Consumo de memoria para size 500.000 según Runtime

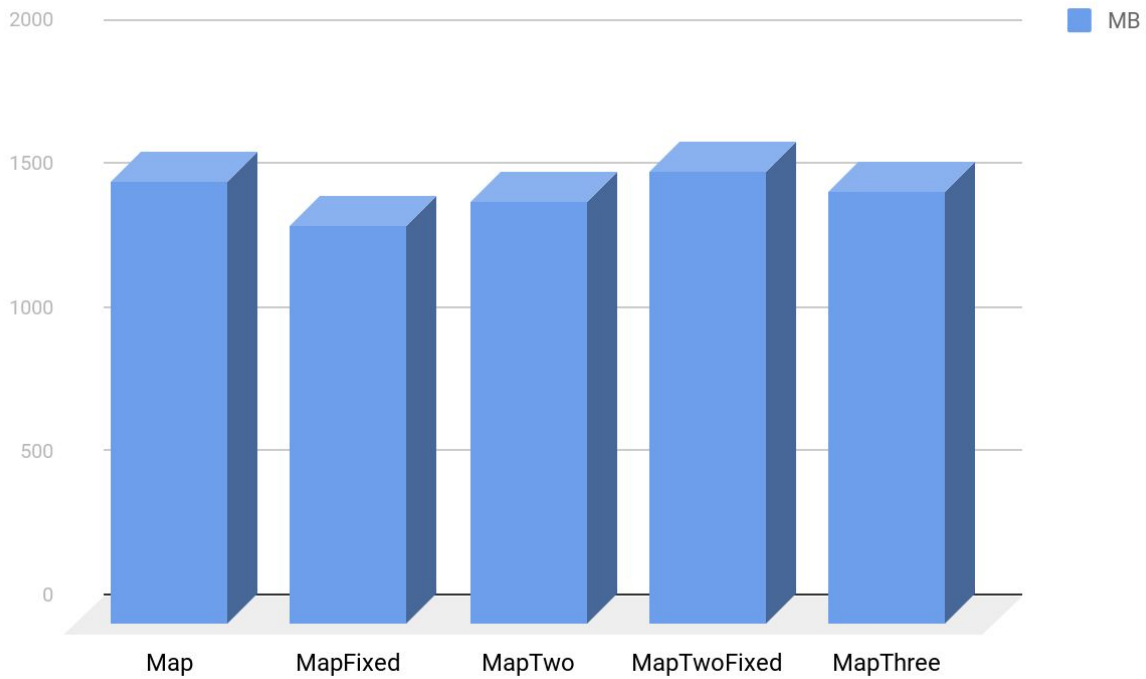


### Tiempo promedio con size 500.000

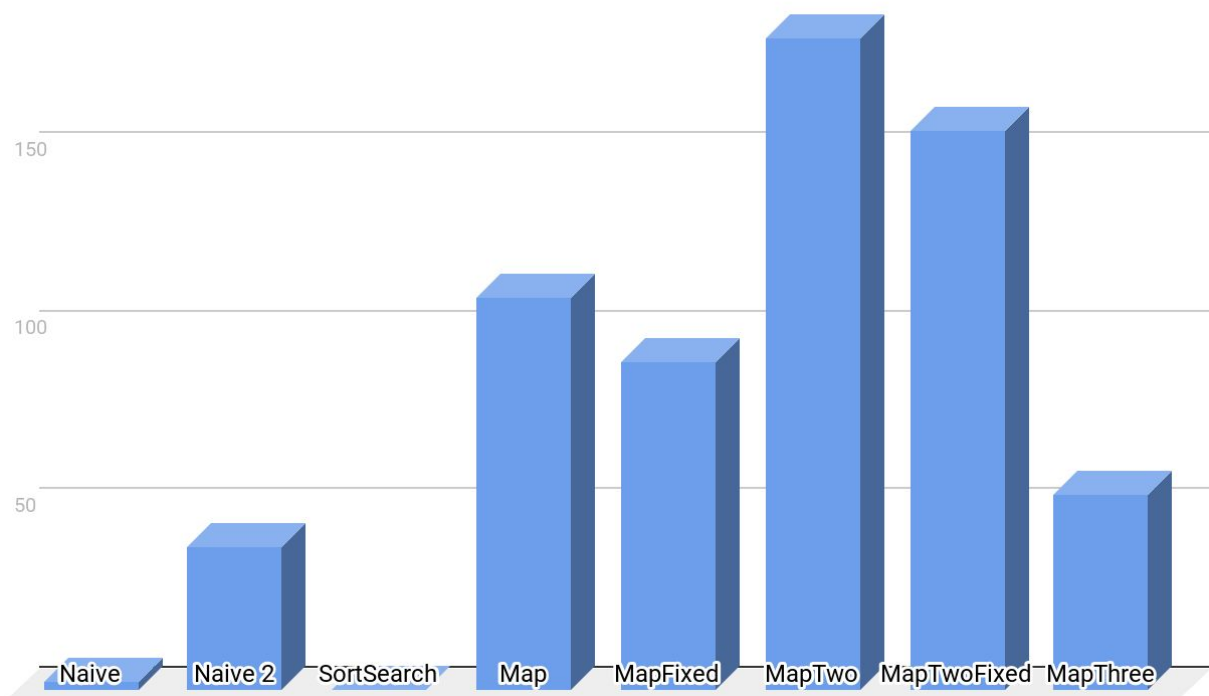
Tiempo promedio con size 500.000



### Consumo de memoria para size 1.000.000 según JVM

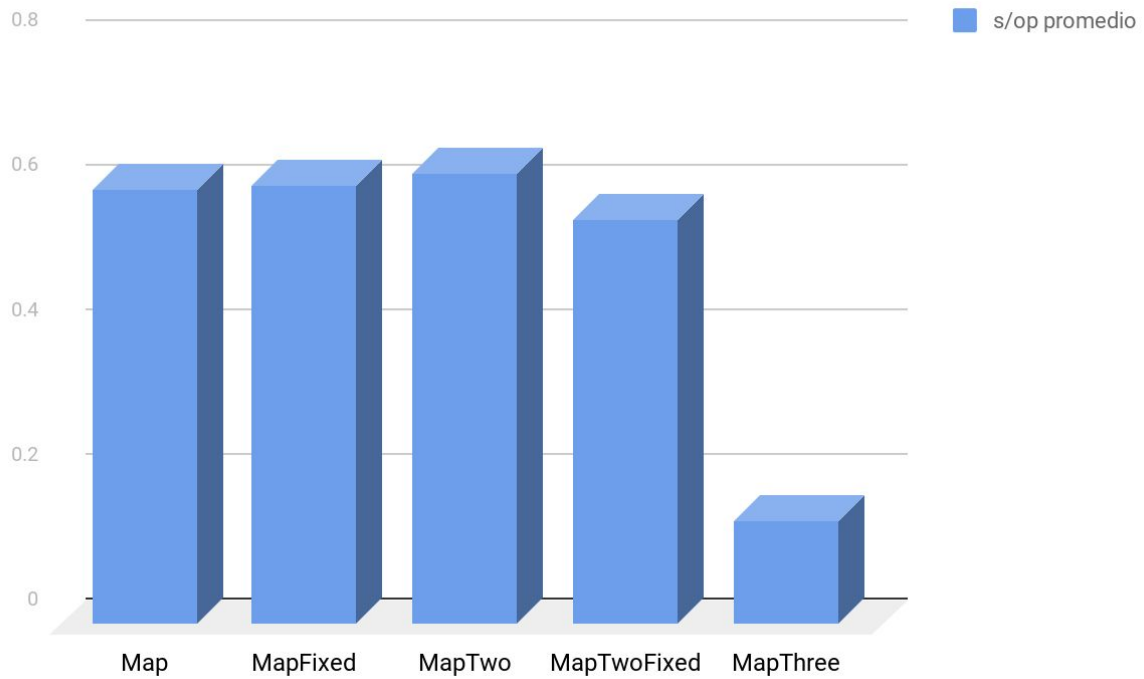


### Consumo de memoria según Runtime para size 1.000.000

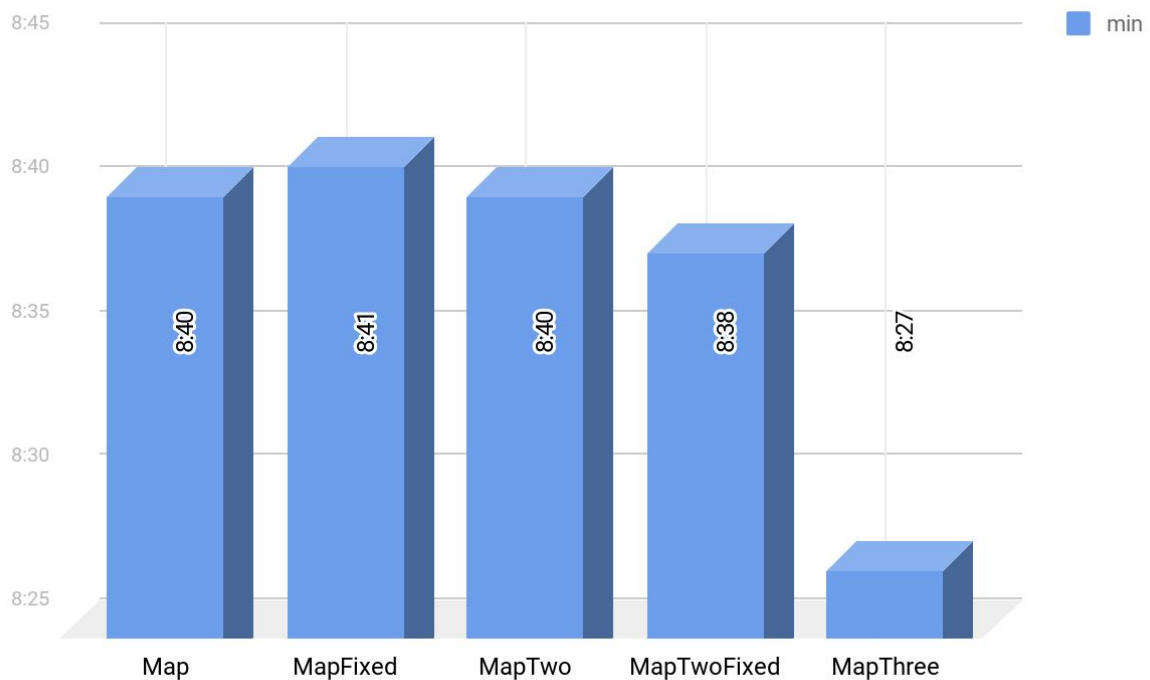




### Tiempo promedio con size 1.000.000

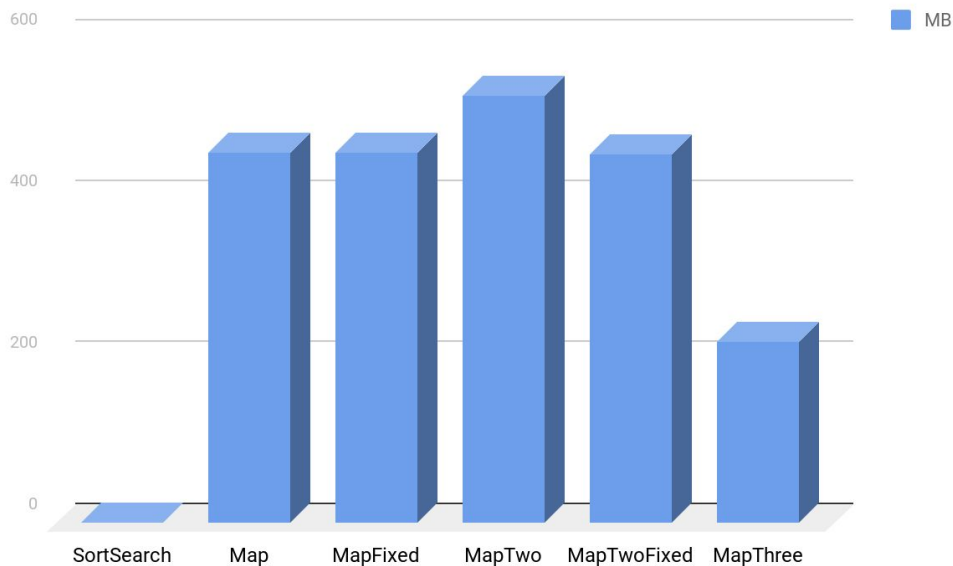


### Tiempo total para size 1.000.000

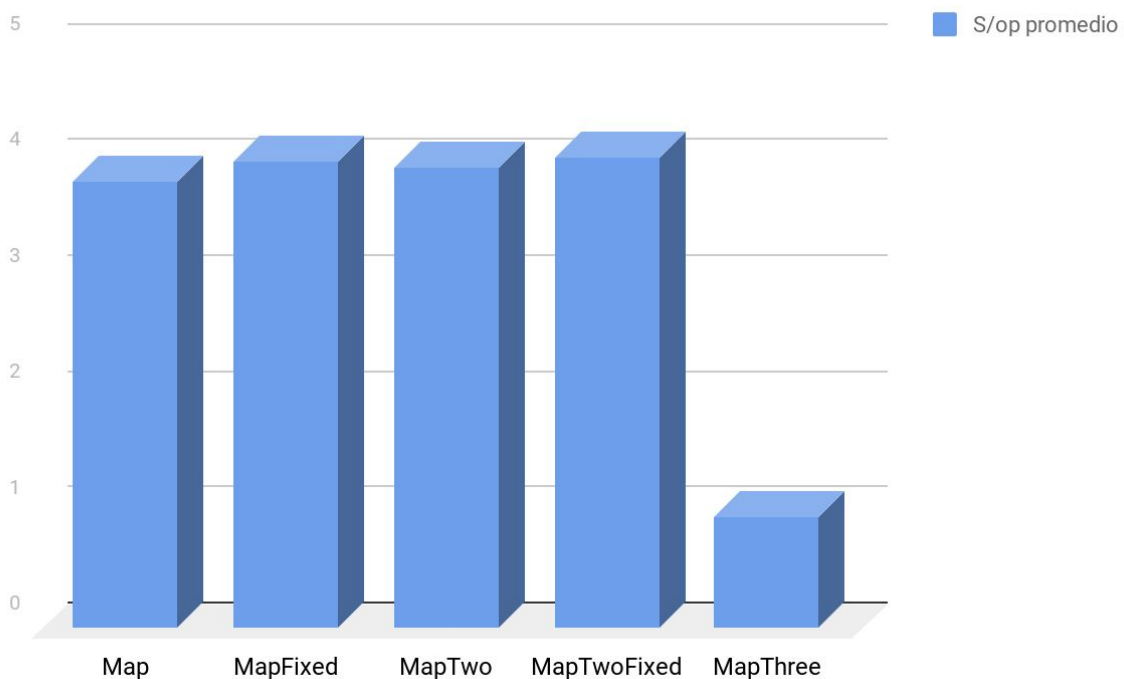


Analizando los resultados obtenidos para size 1.000.000 se decidió, al igual que se realizó en la entrega anterior 100.000, aumentar el número de elementos de prueba para aquellos algoritmos que resultaron mejores. Por esto, se tomaron como referencia los algoritmos que utilizan mapa, y se continuará de ahora en más analizando únicamente las versiones 1 y 2 con las modificaciones introducidas para esta entrega además de la versión 3 que incorpora el uso de Trove.

### Consumo de memoria para size 5.000.000 según Runtime:



### Tiempo promedio con size 5.000.000:



## Conclusiones

### Conclusión anterior

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.

### Conclusión final

Analizando los nuevos valores obtenidos a partir de las nuevas pruebas y de los nuevos algoritmos, podemos concluir que la mejor alternativa es MapThree, la solución se destaca notablemente frente a las demás en todas las pruebas, ya sea con problemas de tamaño 100000, 500000, 1000000 o 5000000, respecto al tiempo y a la memoria usada. Aunque en este último es peor que Naive y Sort, temporalmente es mucho mejor. Creemos que el uso de Trove para el uso de primitivos en las colecciones impacta fuertemente sobre el desempeño del algoritmo, reduciendo el consumo de memoria y considerablemente el tiempo de ejecución, además de la búsqueda de eficiencia en todos los aspectos del código implementado extraídos de MapTwoFixed.