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

Índice

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:

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

Versión 1

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.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): $n / 2^{\wedge m} < 1$. Transformando esta fórmula a un logaritmo en base 2, tenemos que: n < 2^m, por ende log n < m.

Espacio Auxiliar : O(n)

Boulanger, Magalí
Corino, Joshua Yoel

Versión 2

En la entrega anterior habíamos decidido no arreglar el algoritmo, ya que tenía velocidad similar a las versiones usando map y creíamos que con el añadido de la lógica extra para que el algoritmo funcione correctamente se convertiría en una opción no tan buena, pero como no lo habíamos intentado esta vez hicimos la implementación, nombrada en el código como SortSearchFixed:

```
public List<Pair> isSumIn(int[] data, int sum) {
    List<Pair> pairs = new ArrayList<>();
    Arrays.sort(data);
    for (int i=0;i<data.length;i++){
        int index=Arrays.binarySearch(data,sum-data[i]);
        if(index>=0 && index>i){
            Pair par=new Pair(data[i],sum-data[i]);
            pairs.add(par);
            int cursor= index-1;
            if(cursor>i) {
                while (data[cursor] == sum - data[i]) {
                    pairs.add(par);
                    if (cursor - 1 > i)
                        cursor--;
                    else
                        break;
                }
            }
            cursor=index+1;
            if(cursor<data.length) {
                while (data[cursor] == sum - data[i]) {
                    pairs.add(par);
                    if (cursor + 1 < data.length)
                        cursor++;
                    else
                        break;
                }
            }
        }
        else{
            if(index>=0) {
                int cursor = index + 1;
                Pair par = new Pair(data[i], sum - data[i]);
                while (data[cursor] == sum - data[i]) {
                    if (cursor > i) {
                        pairs.add(par);
                    }
```

```
                    if (cursor + 1 < data.length)
                        cursor++;
                    else
                        break;
                }
            }
        }
    }
    return pairs;
}
```

Como se puede observar se hizo la modificación chequeando los valores vecinos al retornado por la búsqueda binaria, ya que pueden ser más de una posición con el valor necesario y, en caso de que sean iguales, incluir la cantidad de pares repetidos en la solución. Del index retornado por la búsqueda nos movemos para atrás y para adelante mirando los vecinos siempre y cuando estemos parados más adelante del i actual (si tendríamos coincidencia más atrás ya la hubiésemos agregado) y antes del final del arreglo.

Complejidad Temporal:

A diferencia de la versión 1, en ésta, dentro del for(n), además de realizar la búsqueda binaria (logn), nos movemos hacia atrás y adelante buscando vecinos (n-constante) por esta razón, la complejidad temporal Big Oh pasa a ser $O(n^2)$.

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, en el código llamada *SolutionHibrid*. Se intentó tener una mejora en el tiempo de ejecución total usando map, 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:

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 *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:

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> 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:

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> frecuencyMap = new HashMap<>();
    int n=data.length;
    // Save elements in map
    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);
    }
    // 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(frecuencyMap.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 < 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;
}
```

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:

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 descripto 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> 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++)
    {
        if(frecuencyMap.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)+
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(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).

Boulanger, Magalí
Corino, Joshua Yoel

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. Asi, 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 frecuencyMap= new TIntIntHashMap();
    TObjectIntHashMap pairMap = new TObjectIntHashMap();
    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]) !=
frecuencyMap.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 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 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 benchmarquea, donde se llama al problem solver declarado en el state para resolver el problema.

Los benchs 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 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
- SortSearchFixed: 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
- SortSearch: 123 MB
- Map: 1300 MB
- MapFixed: 1471 MB
- MapTwo: 1470 MB
- MapTwoFixed: 1470 MB
- MapThree: 270 MB

Size 1.000.000:

- SortSearch: 124 MB
- Map: 1541 MB
- MapFixed: 1383 MB
- MapTwo: 1473 MB
- MapTwoFixed: 1574 MB
- MapThree: 1340 MB

Size 5.000.000:

- SortSearch: 124 MB
- 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 la medición de memoria debía ser totalmente independiente de las soluciones, es decir, sin modificar su código, implementamos la clase *MemoryTest*. En esta clase tenemos como atributo una solución y un método llamado *countMemory()* donde pasamos el código que antes teníamos en cada solución:

```
public void countMemory(ProblemGen problem, int random){
    int [] data=problem.getData();
    Runtime runtime = Runtime.getRuntime();
    long actualMemory =(runtime.totalMemory() - runtime.freeMemory())/
(1024*1024) ;
    solution.isSumIn(data, random);
```

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

Por otro lado, modificamos también el código en *Solutions* que ahora se encarga de crear y llamar a cada uno de los *MemoryTest,* crear el problema, crear el numero random y sugerir a Java usar el garbage collector*:*

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

      ProblemGen problemGen = new ProblemGen();
      ArrayList<MemoryTest> solutors = new ArrayList<MemoryTest>();
      solutors.add(new MemoryTest(new SolutionMap()));
      solutors.add(new MemoryTest(new SolutionMapFixed()));
      solutors.add(new MemoryTest(new SolutionMapTwo()));
      solutors.add(new MemoryTest(new SolutionMapTwoFixed()));
      solutors.add(new MemoryTest(new SolutionMapThree()));
      solutors.add(new MemoryTest(new SolutionNaive()));
      solutors.add(new MemoryTest(new SolutionNaive2()));
      solutors.add(new MemoryTest(new SolutionSortSearch()));
      problemGen.genRandomProblem(100000);
      int random=(int)(Math.random() * 2 * Integer.MAX_VALUE +
Integer.MIN_VALUE/2);
      for (MemoryTest i: solutors){
            i.countMemory(problemGen, random);
            Runtime.getRuntime().gc();
      }
}
```

Se volvieron a hacer las pruebas con problemas de tamaño 100000, 500000, 1000000 y 5000000. Se añadieron con tamaño 10000000 y los resultados obtenidos fueron los siguientes:

Algoritmo \ Size	**100.000**	**500.000**	**1.000.000**	**5.000.000**	**10.000.000**
Naive	1 MB	1 MB	2 MB	----------	----------
Naive 2	26 MB	54 MB	40 MB	----------	----------
SortSearch	0 MB	0 MB	0 MB	----------	----------
SortSearchFixed	0 MB	0 MB	3 MB	5 MB	11 MB

Map	14 MB	38 MB	110 MB	460 MB	646 MB
MapFixed	13 MB	37 MB	92 MB	459 MB	659 MB
MapTwo	14 MB	73 MB	183 MB	529 MB	1047 MB
MapTwoFixed	13 MB	70 MB	157 MB	458 MB	777 MB
MapThree	6 MB	28 MB	55 MB	224 MB	460 MB

Tiempos

Tiempos con size 100.000

- Naive:
  - 3,019 ±(99.9%) 0,065 s/op [Average]
- NaiveTwo:
  - 15,038 ±(99.9%) 1,931 s/op [Average]
- SortSearch:
  - 0,013 ±(99.9%) 0,001 s/op [Average]
- SortSearchFixed:
  - 0,013 ±(99.9%) 0,001 s/op [Average]
- Map:
  - 0,020 ±(99.9%) 0,001 s/op [Average]
- MapFixed:
  - 0,018 ±(99.9%) 0,001 s/op [Average]
- MapTwo:
  - 0,018 ±(99.9%) 0,001 s/op [Average]
- MapTwoFixed:
  - 0,015 ±(99.9%) 0,001 s/op [Average]
- MapThree:
  - 0,013 ±(99.9%) 0,001 s/op [Average]

Tiempos con size 500.000:

Considerando los resultados en los tiempos con size 100000 elegimos seguir haciendo las pruebas con los algoritmos con mejores tiempos, descartando así Naive y Naive2 por su gran diferencia temporal con los demás.

- SortSearchFixed:
  - 0,017 ±(99.9%) 0,003 s/op [Average]
- Map:
  - 0,26 ±(99.9%) 0,042 s/op [Average]
- MapFixed:
  - 0,249 ±(99.9%) 0,009 s/op [Average]
- MapTwo:
  - 0,245 ±(99.9%) 0,006 s/op [Average]
- MapTwoFixed:
  - 0,242 ±(99.9%) 0,014 s/op [Average]
- MapThree:
  - 0,073 ±(99.9%) 0,003 s/op [Average]

Tiempos con size 1.000.000

- SolutionSortSearch:
  - 0,036 ±(99.9%) 0,007 s/op [Average]
- Map:
  - 0,620 ±(99.9%) 0,017 s/op [Average]
- MapFixed:
  - 0,618 ±(99.9%) 0,035 s/op [Average]
- MapTwo:
  - 0,641 ±(99.9%) 0,020 s/op [Average]
- MapTwoFixed:
  - 0,600 ± (99.9%) 0,025 s/op [Average]
- MapThree:
  - 0,163 ± (99.9%) 0,005 s/op [Average]

Tiempos con size 5.000.000

- SortSearchFixed:
  - 0,154 ±(99.9%) 0,024 s/op [Average]
- Map:
  - 3,918 ±(99.9%) 0,589 s/op [Average]
- MapFixed:
  - 4,156 ±(99.9%) 0,418 s/op [Average]
- MapTwo:
  - 4,052 ±(99.9%) 0,846 s/op [Average]
- MapTwoFixed:
  - 4,181 ±(99.9%) 0,373 s/op [Average]
- MapThree:

- 1,012 ±(99.9%) 0,019 s/op [Average]

Tiempos con size 10.000.000

Si bien con el size 5000000 se vieron grandes diferencias entre MapThree, SortSearchFixed y los demás, decidimos hacer una prueba con un size más grande para ver como se comportan con un nivel mas de magnitud. Los resultados fueron los siguientes:

- SortSearchFixed:
  - 0,445 ±(99.9%) 0,232 s/op [Average]
- Map:
  - 9,426 ±(99.9%) 0,971 s/op [Average]
- MapFixed:
  - 9,709 ±(99.9%) 1,024 s/op [Average]
- MapTwo:
  - 11,297 ±(99.9%) 0,965 s/op [Average]
- MapTwoFixed:
  - 10,782 ±(99.9%) 0,757 s/op [Average]
- MapThree:
  - 2,197 ±(99.9%) 0,294 s/op [Average]

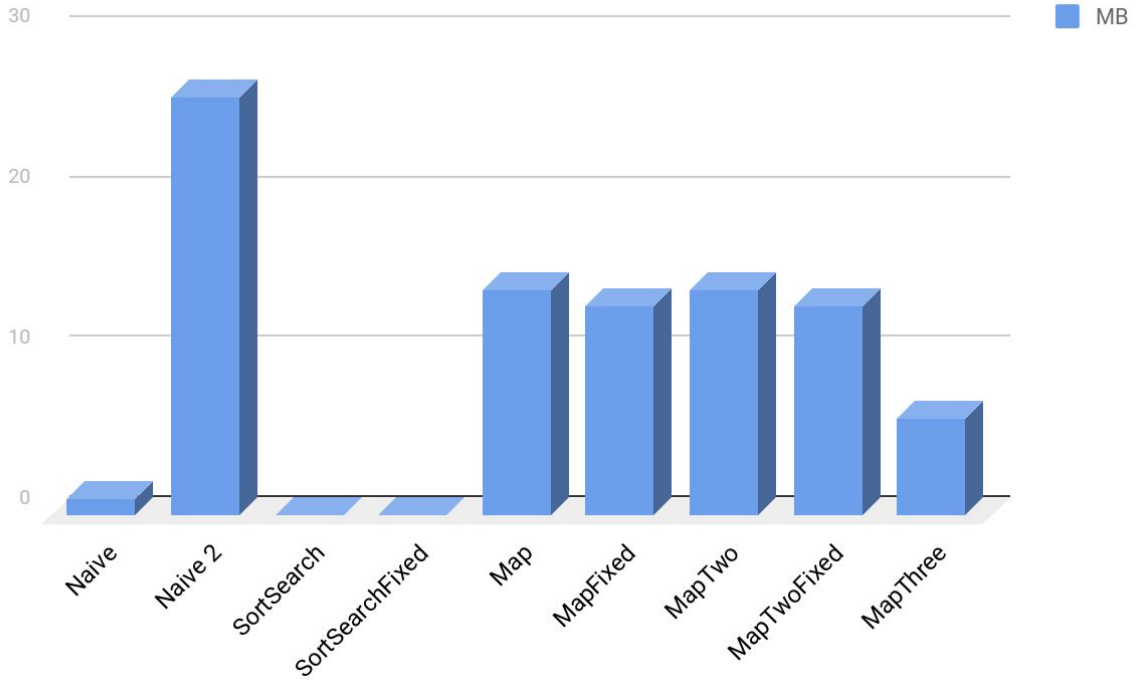
Tiempos con size 100.000.000

Para hacer una última comparación temporal probamos con un problema de cien millones comparando los dos mejores algoritmos (MapThree y SortSearchFixed). Resultados:

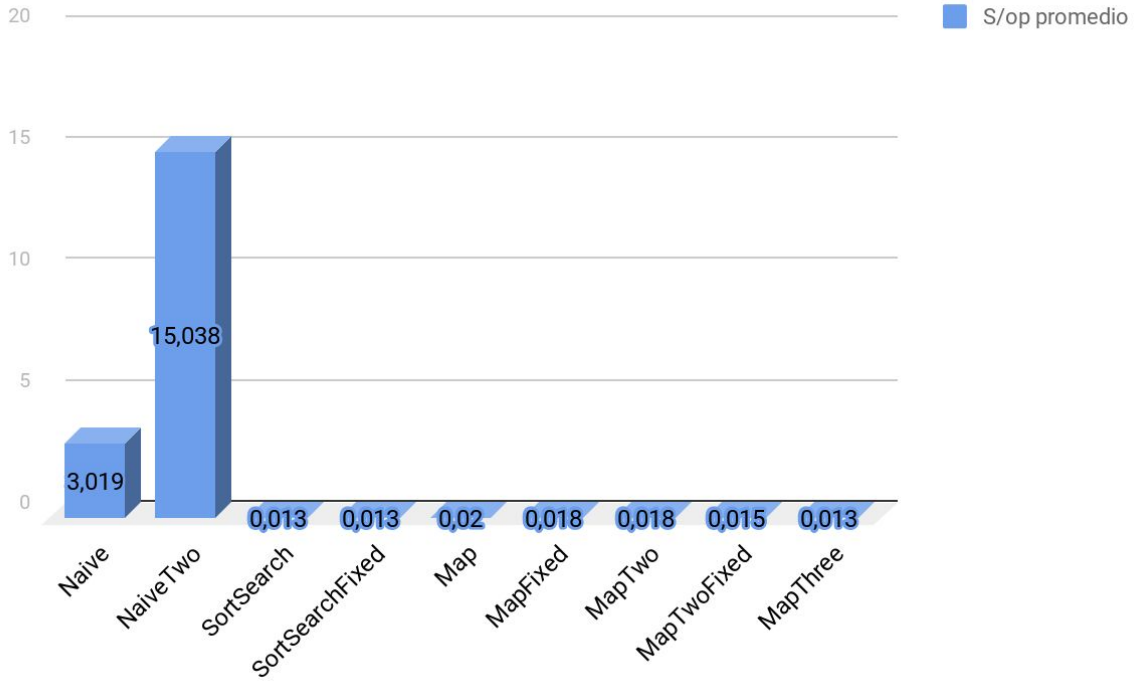
- SortSearchFixed:
  - 4,728 ±(99.9%) 1,916 s/op [Average]
- MapThree:
  - 28,513 ±(99.9%) 2,777 s/op [Average]

Gráficos

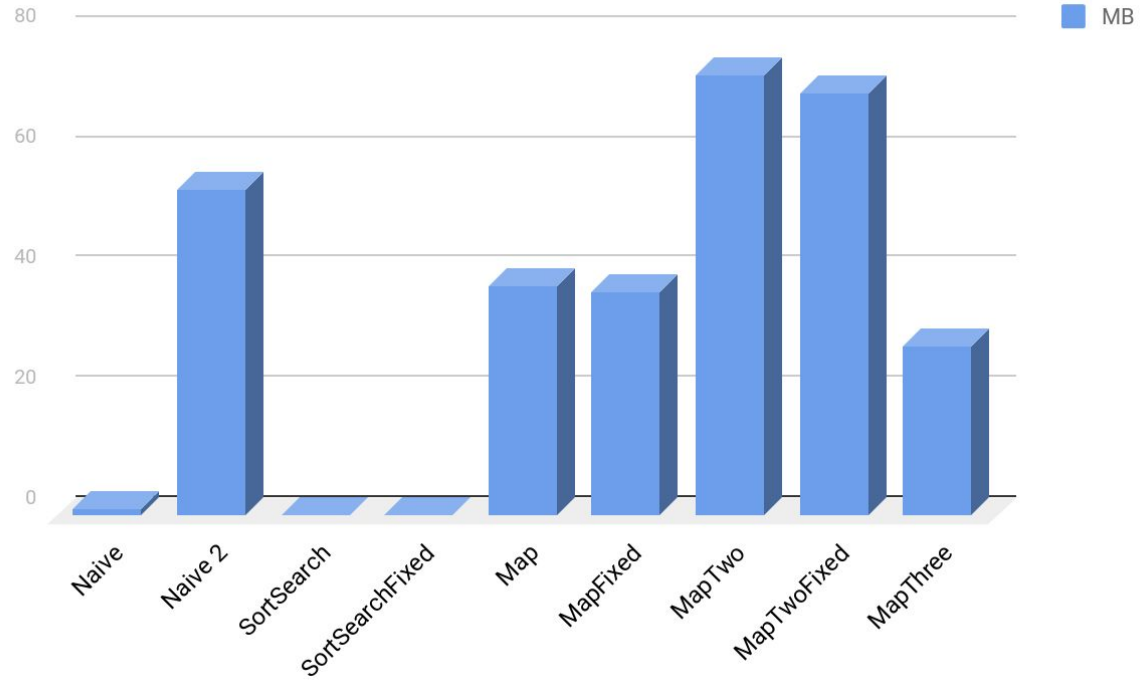
Consumo de memoria para size 100.000 según Runtime



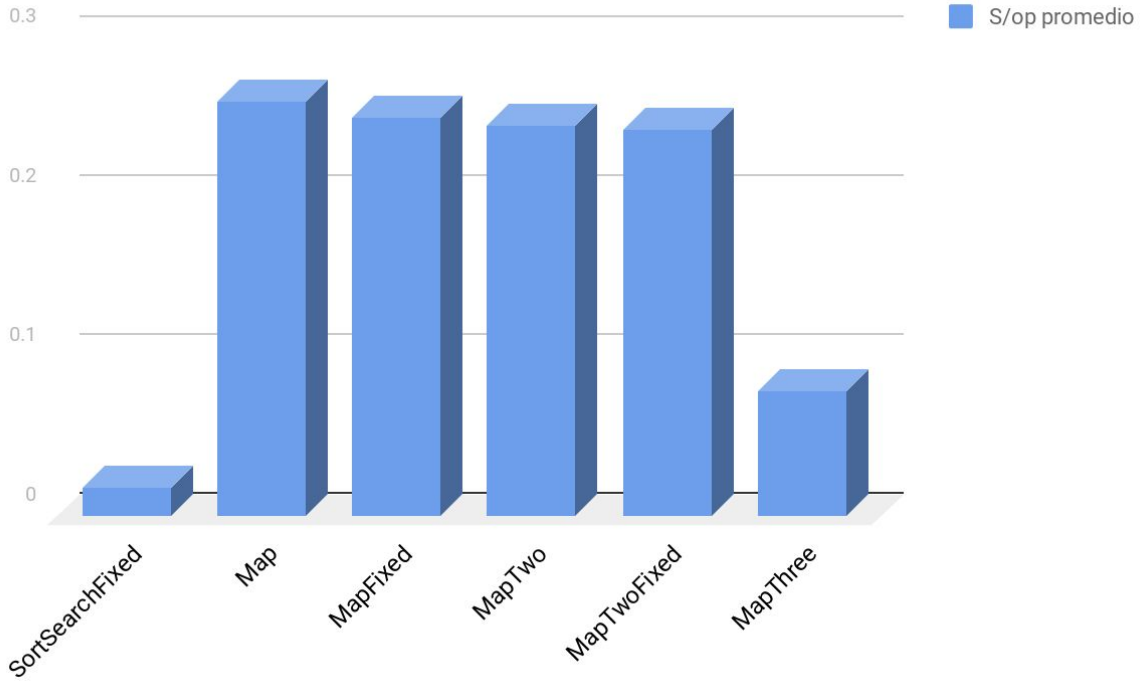
Tiempo promedio con size 100.000



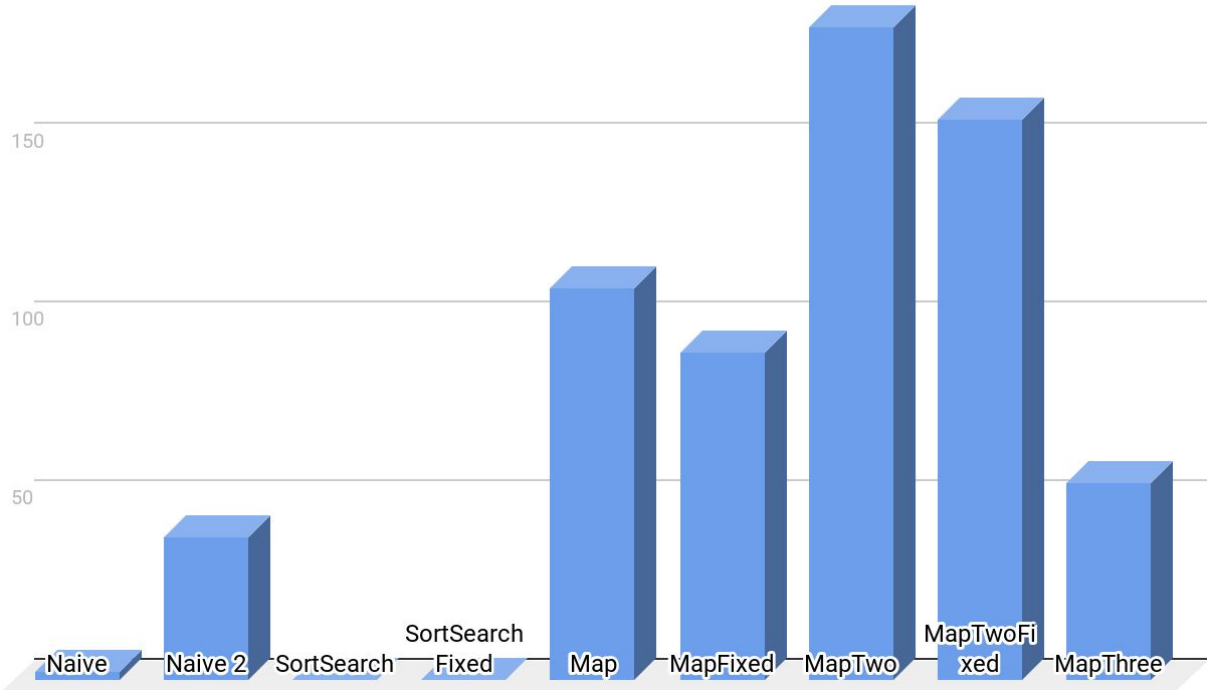
Consumo de memoria para size 500.000 según Runtime



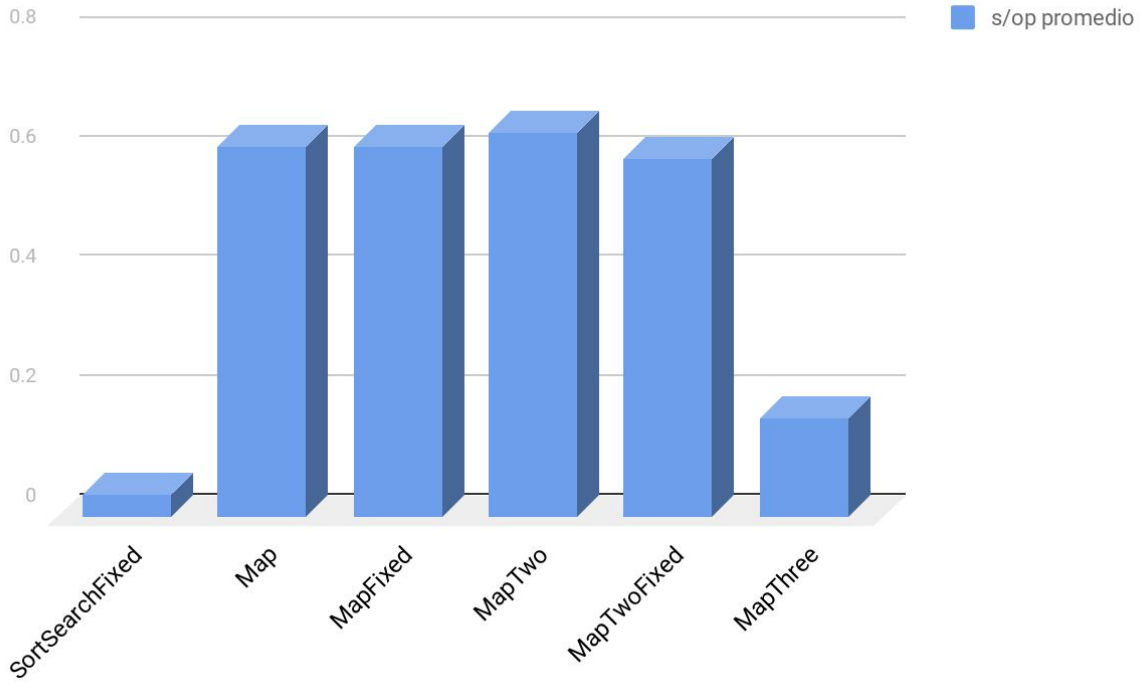
Tiempo promedio con size 500.000



Consumo de memoria según Runtime para size 1.000.000



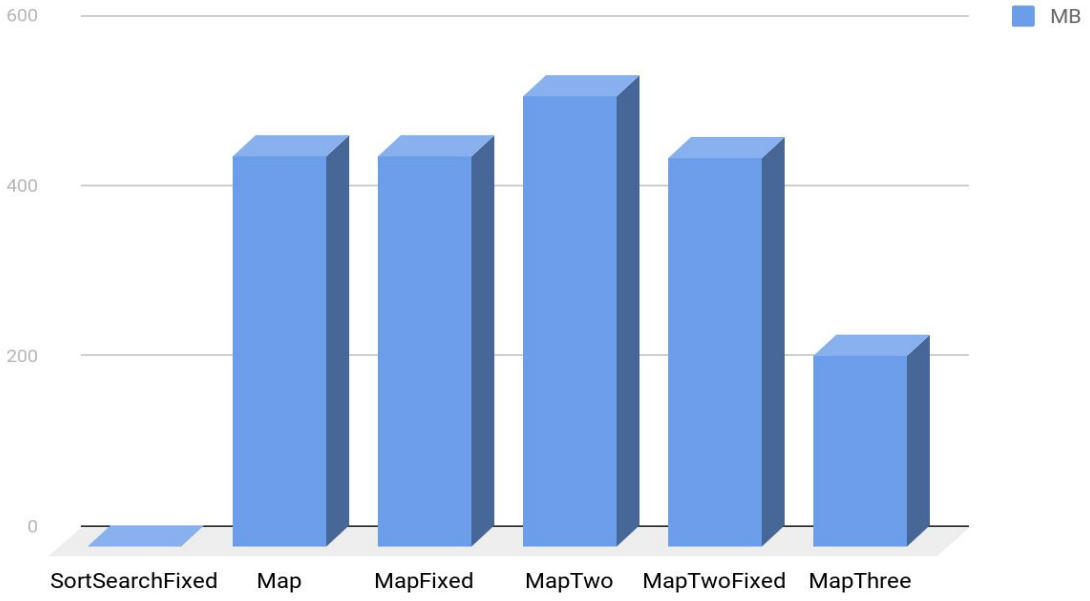
Tiempo promedio con size 1.000.000



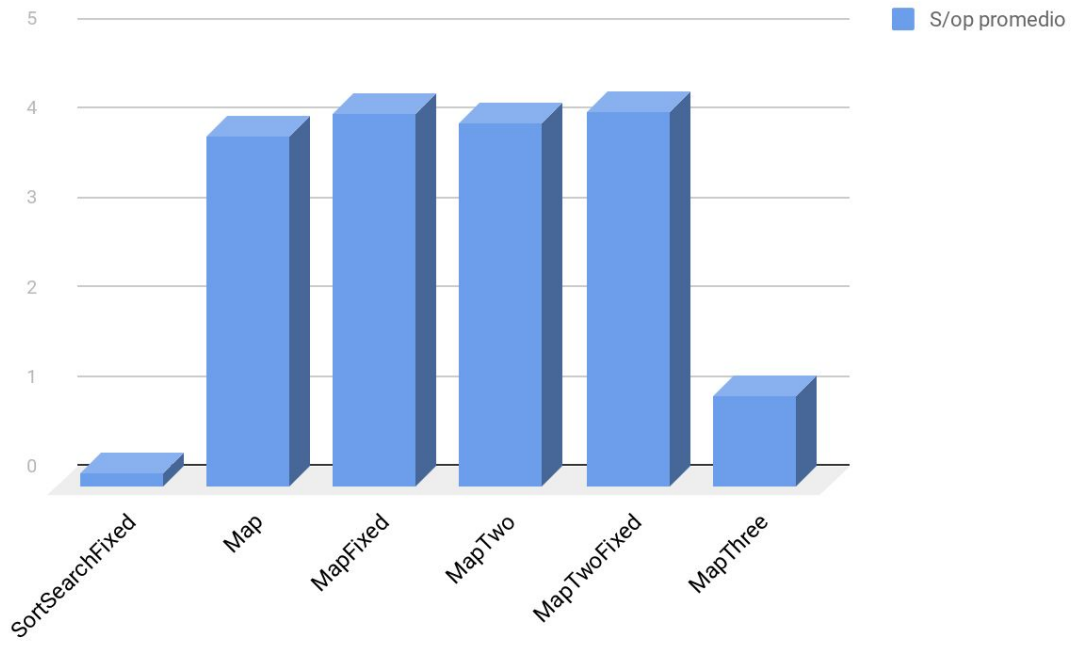
Analizando los resultados obtenidos para size 1.000.000 se decidió, al igual que se realizó en la entrega anterior 100.000, aumentar el número de elementos de prueba para aquellos algoritmos que resultaron mejores. Por esto, se tomaron como referencia los algoritmos que utilizan mapa, y se continuará 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.

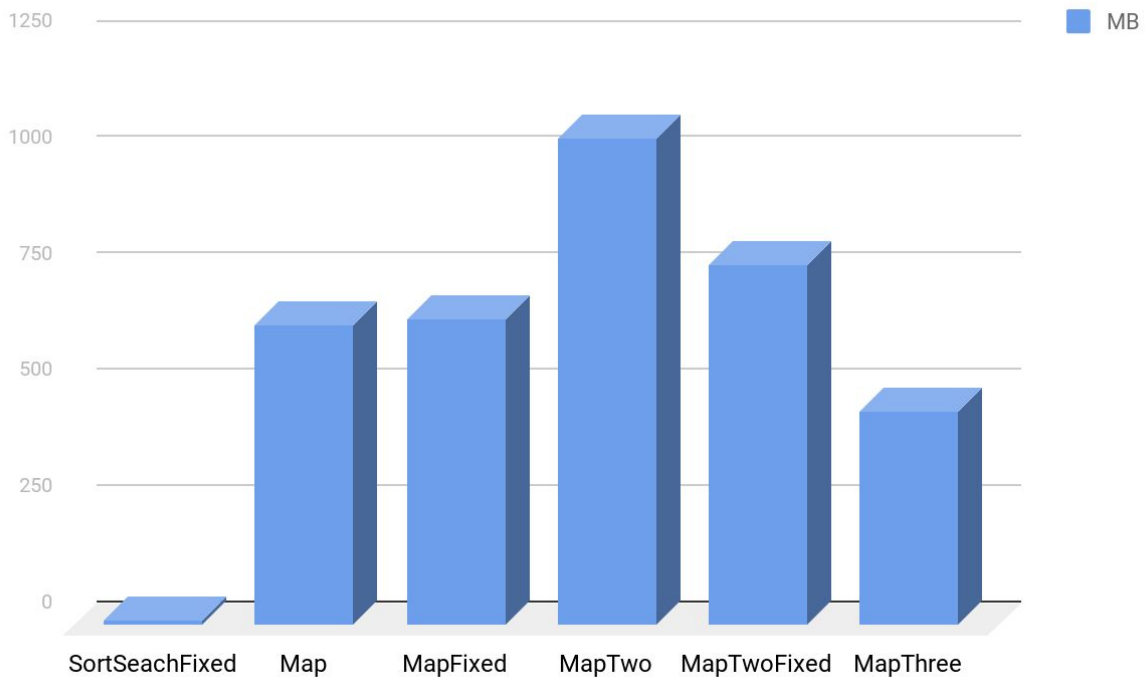
Consumo de memoria para size 5.000.000 según Runtime



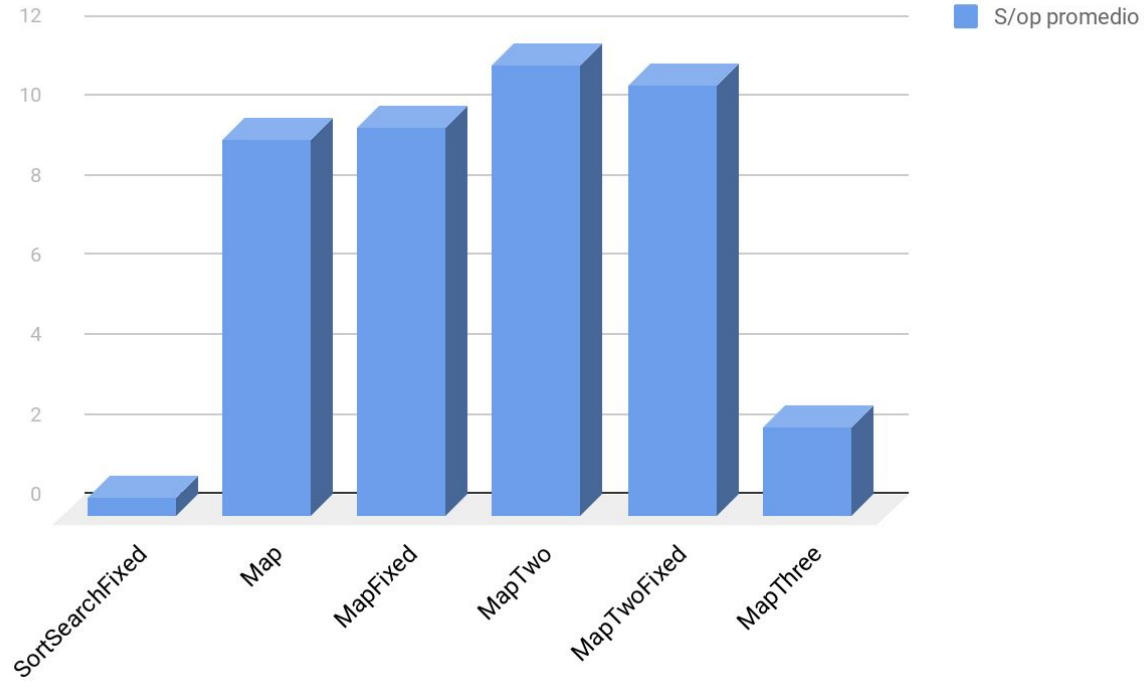
Tiempo promedio con size 5.000.000



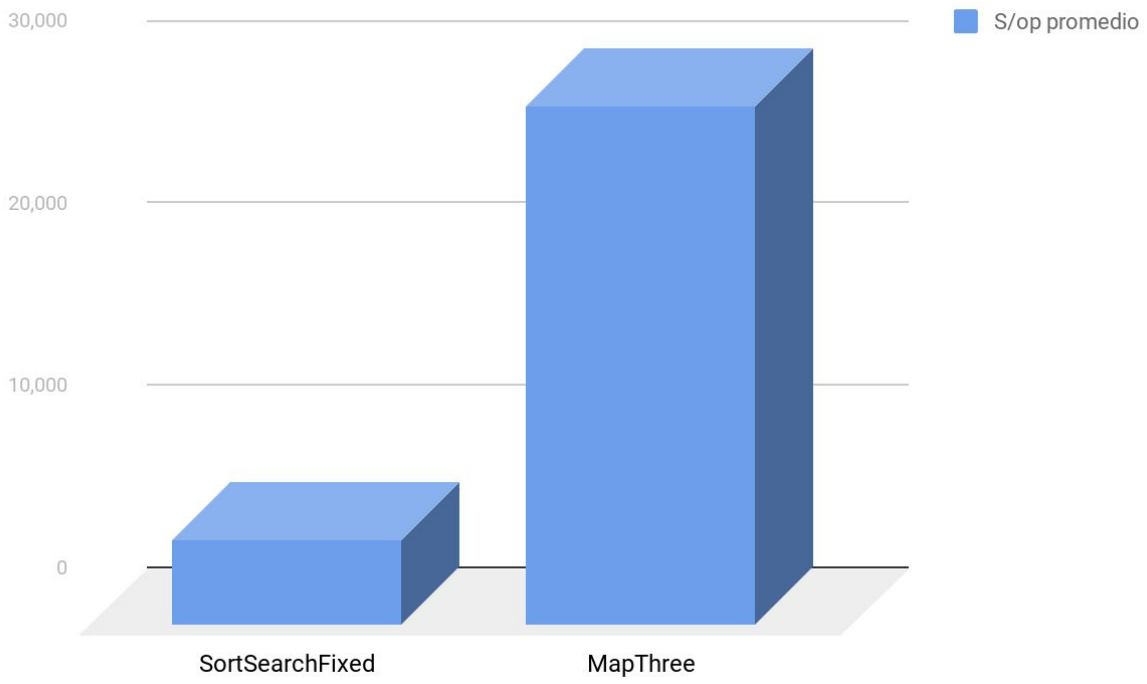
Consumo de memoria para size 10.000.000 según Runtime



Tiempo promedio con size 10.000.000



Tiempo promedio con size 100.000.000



Conclusiones

Conclusión uno

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 dos

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.

Conclusión final

Haciendo nuevamente análisis de los resultados obtenidos encontramos como mejor el último algoritmo implementado *SearchSortFixed*, en segundo lugar *MapThree*, luego *Map* y *MapTwoFixed* que tienen un rendimiento y uso de memoria similar, luego MapTwo que tiene igual rendimiento que los dos anteriores pero ocupa mas memoria y por último *Naive* y *Naive2* que si bien consumen poca memoria son demasiado lentos como para ser considerados.

SearchSortFixed es el mejor respecto a espacio ya que no usa estructuras auxiliares para resolver el problema (y *SearchSort* es descartado porque no resuelve el problema), los pocos MB que ocupa son los objetos de tipo Par creados. Temporalmente es el mejor porque si bien la complejidad temporal está acotada superiormente en Big Oh por n^2 esto a la práctica no es así y en promedio termina estando más cerca de n.log(n). Estas dos razones conllevan que el algoritmo escale muy bien las magnitudes y supere a las demás implementaciones como se puede observar fácilmente en todos los gráficos.

Correcciones

- La SolutionHibrid sigue retornando 9 pares para el arreglo {2, 2, 2} y el target 4, cuando debería devolver 3 pares.
- Las soluciones SolutionSortSearch y SolutionSortSearchFixed no retorna ningún par si el arreglo es {2,2} y el target es 4.
- La SolutionSortSearchFixed arroja un ArrayIndexOutOfBoundException si el target es 4 y el arreglo es {1,2,3,4,5,2,0, 0,-1,2}.
- El código reportado en el informe para SolutionSortSearchFixed no es el mismo que en el archivo .java.
- Revisen la semántica del código, ya que hay partes que parece redundante.