Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

YUNUS ÇEVİK
141044080

Course Assistant: Fatma Nur Esirci

1 Double Hashing Map

1.1 Pseudocode and Explanation

1.1.1 DoubleHashMap.find(Object key)

1. Set index to key.hashCode() % dhmTable.length.
2. İf index is negative, add dhmTable.length.
3. While dhmTable[index] is not empty and the key is not at dhmTable[index]
4. Increment index.
5. if index is greater than or equal to dhmTable.length
6. Set index to 0.
7. Return the index.

find(Object key) => Parametre olarak gelen key değerini tablo içerisinde aratarak kaçıncı indekse sahip olduğunu veren metottur. Bu metotta gelen key değerinin tablo üzerinde nerede olduğunu bilmek için key.hashCode() metotu çağrılır ve tablonun boyutuna bölünerek indeks değeri hesaplanır ve tablo da bu index değerine gidilerek aradığımız eleman var ise bu indeks değeri döndürülür.

1.1.2 DoubleHashMap.get(Object key)

1. Find the first dhmTable element that is empty ot the dhmTable element that contains the key.
2. if the dhmTable element found contains the key
3. Return the value at this dhmTable element.
4. else
5. Return null.

get(Object key)=> Find metodu çağrılarak aranan key değerinin kaçıncı indekste olduğu tespit edilir. Daha sonra bu indeks değerine tablo üzerinden erişilerek aranan key değeri ile aynı key değeri saptanır ise value bilgisi döndürülür. Eğer key değerleri bir biri ile uyuşmuyor ise null döndürülür.

1.1.3 DoubleHashMap.hash1(K key)

1. Return Math.abs(key.hashCode() % dhmTable.length)

hash1(K key)=> Key değerinin hashCode() metodunun çağrılmış halinin tablo boyutuna göre modu alındığı takdirde key değerinin tabloda hangi yerde olduğunu gösteren indeks değerini verir.

1.1.4 DoubleHashMap.hash2(K key)

1. Return Math.abs(PRIME - (key.hashCode() % PRIME))

hash2(K key)=> Belirlenen sabit bir PRIME değerinden, key değerinin hashCode() metodunun çağrılmış halinin tablo boyutuna göre modunun alınmış hali çıkarılarak yeni bir indeks değeri belirlenir. Bu olay collision olduğu takdirde yapılır ve double hash mantığı bu şekilde gerçekleşir ve collisionlar engellenmeye çalışılır.

1.1.5 DoubleHashMap.put(K key, V value)

1. Set index to hash1 metod.
2. if dhmTable[index] is not empty
3. if dhmTable[index].key equal to key
4. Set oldVal to dhmTable[index].value
5. Set dhmTable[index] to new Pair< key, value>
6. Return the old value
7. else
8. Set index2 to hash2 metod
9. if an empty element was found //(dhmTable[index] is empty)
10. Set dhmTable[index2] to new Pair< key, value> and Increment size.
11. Check for need to rehash.
12. Return null.
13. else
14. Set i to 1
15. While condition is true
16. Set newIndex to (index + i * index2) % dhmTable.length
17. if an empty element was found //(dhmTable[index] is empty)
18. Set dhmTable[newIndex] to new Pair< key, value> and Increment size.
19. Check for need to rehash.
20. Return null.
21. Increment i
22. else
23. Set dhmTable[newIndex] to new Pair< key, value> and Increment size.
24. Return dhmTable[newIndex].value

put(K key, V value)=> Key ve value bilgilerini tabloya ekleyebilmek için ilk başta hash1 metodu uygulanarak indeks değeri belirlenir ve bu indeks eğer boş ise key ve value bilgisi pair olarak eklenir. Ancak bu indekse sahip bir değer varsa ve key değerleri aynı ise value bilgisi güncellenir. Eğer indeks değerinde her hangi bir key değeri var ise collision olduğundan dolayı hash2 metodu uygulanır ve collision durumu önlenmeye çalışılır. Böyle collision olmadan tablo üzerinde ekleme işlemi gerçekleştirilir.

1.1.6 DoubleHashMap. rehash()

1. Allocate a new hash table is double the size and has an odd length.
2. Reset the number of keys and number of deletions to 0.
3. Reinsert each dhmTable that has not been deleted in the new hash table.

rehash()=> Tablo üzerinde eklenecek bir indeks kalmadığı takdirde bu rehash işlemi yapılarak table boyutu iki katının bir fazlasına çıkarılır. Daha sonra eski elemanlar yeni hash tabloya göre tekrardan put metodu ile eklenir.

1.1.7 DoubleHashMap. remove(Object key)

1. Find the first table element that is empty or the dhmTable element that contains the key.
2. if an empty element was found (dhmTable[index] is empty)
3. Return null.
4. Key vas found. Remove this dhmTable element bu setting it to referance Pair<>(null, null), increment deleteSize, and decrement size.
5. Return the value associated with this key.

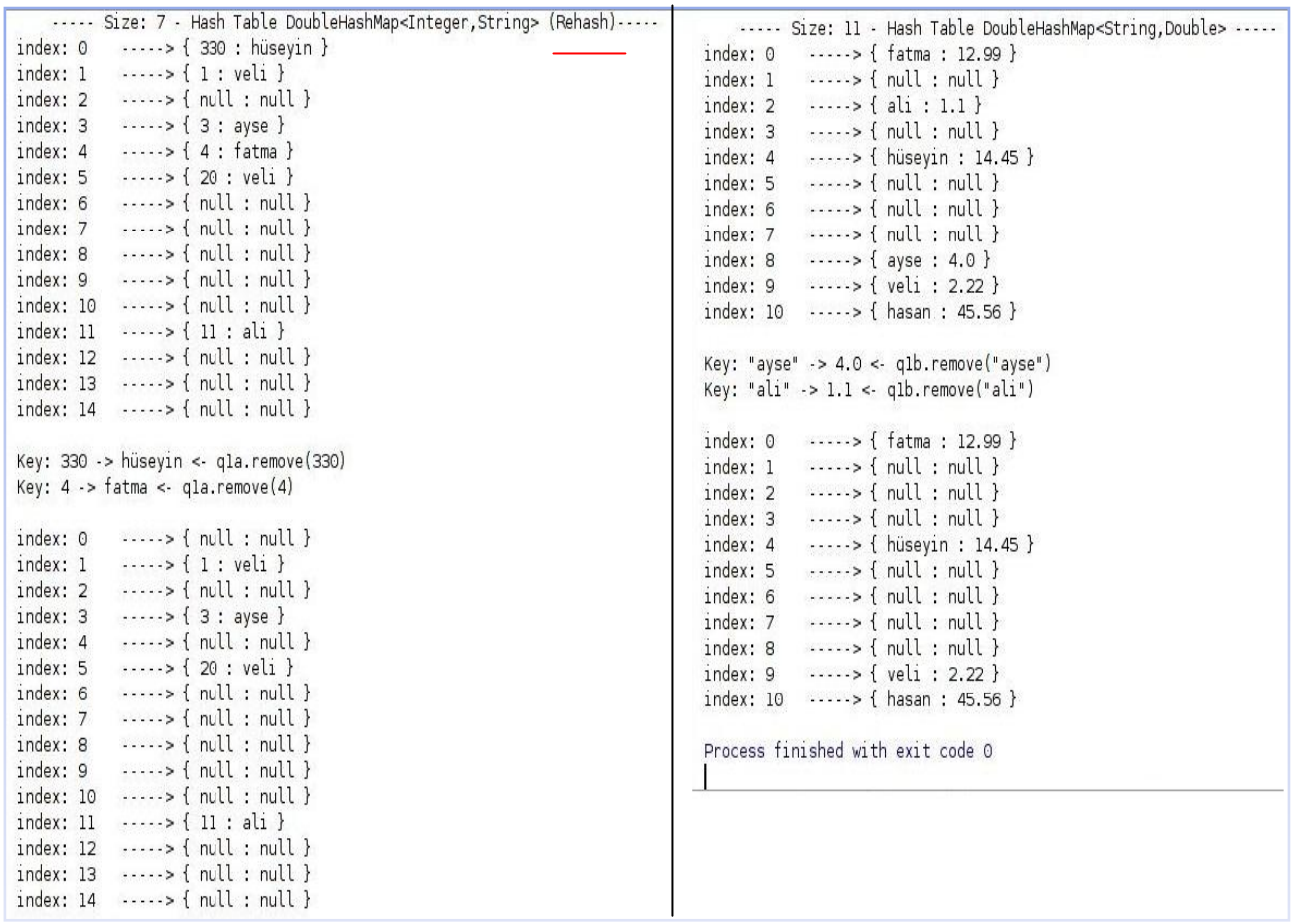
remove(Object key)=> Key değerinin belirlediği indeks değerine tablo üzerinden ulaşılarak bu yerdeki bilgi yerine null atanarak silme işlemi gerçekleştirilir. Silinen değer geri döndürülür.

Not: Double Hashing Map yapısını oluşturmak için Java' nın Map interface yapısını implement ederek Map içindeki metotları kullanma imkanı sağladık ayrıca Map yapısını oluşturmamız için inner class olan bir Pair class'ı yazarak key ve value değerlerini elde etmiş olduk. Collision durumu önlemek için yaptığımız bu hash table' nda aynı key değerine sahip bir başka veri geldiği taktirde güncelleme yapılacaktır. Ancak key değerinin hashCode() değerine göre collision olursa ona göre double hash yapılarak collison önlenecektir.

1.2 Test Cases

Double Hashing Map' i test etmek için "hw5.Q1" içerisinde yer alan "Main.java" yı çalıştırmak yeterlidir. Boyutları farklı iki hash tablo üzerinde add, remove, get, size ve isEmpty metodları kullanılmıştır. Collision durumunun oluştuğunu ve daha sonra double hashing uygulayarak tabloda boş bir key değeri göz önünde bulundurularak tablo üzerine yerleştirilmiştir. Aşağıda bulunan ekran çıktısında bu olay gösterilmektedir.

Double Hashing Map Screen Shot



2 Recursive Hashing Set

2.1 Pseudocode and Explanation

2.1.1 RecursiveHashingSet. contains(Object val)

1. Call ontains(EntrySet<E> dataArr [], Object val). //(Helper Metod)
2. Return containsRet.

2.1.1.1 contains(EntrySet<E> dataArr [], Object val)

1. Set containsRet to false.
2. if dataArr is not empty.
3. Set index to hash metod.
4. if dataArr[index].data is not empty and dataArr[index].data equal to value.
5. Set containsRet to true.
6. else
7. Call contains(dataArr[index].dataArr, val) // (Recursive Call)

contains(Object val)=> Parametre olarak aldığı value bilgisi ile kendi içinde helper metot olan overload edilmiş contains metodunu çağırır ve bu metot recurive çağrı ile gelen value bilgisini, Hash Set içinde olup olmadığını True / False olarak döndürür.

2.1.2 RecursiveHashingSet. add(E e)

1. Call add(EntrySet<E> dataArr [], E e).
2. Return addRet.

2.1.2.1 RecursiveHashingSet.add(EntrySet<E> dataArr [], E e)

1. Set addRet to false.
2. if dataArr.length small equal to 1.
3. Set addRet to false.
4. Return.
5. Set index to hash metod
6. if an empty element was found. //(dhmTable[index] is empty)
7. Set dataArr[index] to new EntrySet<>().
8. Set dataArr[index] to value. // (e)
9. Increment dataArr[index].size.
10. Increment size.
11. Set addRet to true.
12. Return.
13. else
14. if dataArr[index].data is not empty and dataArr[index].data equal to value. //(e)
15. Set addRet to false.
16. Return.
17. if dataArr[index].data is not empty and dataArr[index].dataArr is empty.
18. Set tempCap to (dataArr.length -3). // decrement length with 3
19. if tempCap smal equal to 1.

20. Set tempCap to 1.
21. Set dataArr[index].dataArr to new EntrySet[tempCap]
22. Call add(dataArr[index].dataArr, e) // (Recursive Call).
23. else if dataArr[index].data is not empty and dataArr[index].dataArr is not empty.
24. Call add(dataArr[index].dataArr, e) // (Recursive Call).

add(E e)=> Parametrede gelen value bilgisini HashSet Table üzerinde hashCode() değerine göre belirlenen indekse yerleştirmek için yazılmış olan bu metot içerisinde overload edilmiş add metodunu çağırır ve ekleme işlemini gerçekleştirir. Ancak collision durumu var ise tabloda bulunan collision kısmından yeni bir boyutu daha düşük bir tablo açılarak bu kısmında hash metodunun belirlediği indeks değerine göre value yerleştirilir.

2.1.3 RecursiveHashingSet.hash(EntrySet<E> dataArr [], E e)

1. Return Math.abs(e.hashCode() % dataArr.length)

hash1(K key)=> E tipinde bulunan değerinin hashCode() metodunun çağrılmış halinin tablo boyutuna göre modu alındığı takdirde E tipindeki değerin tabloda hangi yerde olduğunu gösteren indeks değerini verir.

2.1.4 RecursiveHashingSet. remove(Object val)

1. Call remove(EntrySet<E> dataArr [], Object val).
2. Return removeRet.

2.1.4.1 remove(EntrySet<E> dataArr [], Object val)

1. Set removeRet to false.
2. if dataArr is not empty.
3. Set index to hash metod.
4. if dataArr[index] is not empty
5. if dataArr[index].data is not empty and dataArr[index].data equal to value.
6. Set dataArr[index].data to null.
7. Decrement size.
8. Set removeRet to true.
9. Return.
10. else
11. Call remove(dataArr[index].dataArr, val). // (Recursive Call)

remove(Object val)=> Parametrede aldığı value değerinin hash metodundan çıkan indeks değeri ile HashSet Table üzerinde gösterilen indeks değerine recursive olarak erişilerek tablo üzerinde silme işlemi gerçekleştirilir yani bu yere null değeri atanır.

2.2 Test Cases

Recursive Hashing Set' i test etmek için "hw5.Q2" içerisinde yer alan "Main.java" yı çalıştırmak yeterlidir. Boyutları farklı iki hash tablo üzerinde add, remove, contains, size ve isEmpty metodları kullanılmıştır. Collision durumu oluştuğu takdirde tabloda hangi indekste collision oluştu ise o indekse ayrıca yeni bir tablo açılarak collision oluşturan değerin oraya göre hash metodundan çıkan indeks ile açılan tabloya eklenmesi sağlanır.

Aşağıda bulunan ekran çıktısında bu olay gösterilmektedir.

Recursive Hashing Set Screen Shot

```
----- Size: 7 - Hash Table RecursiveHashingSet<Integer> -----
Is Empty: false

 index => 5 - true <- q4a.add(12)
 index => 1 - true <- q4a.add(22)
12<- ! COLLISION ! -> 19 Collision Index => 5
 index => 3 - true <- q4a.add(19)
 index => 2 - true <- q4a.add(135)
 index => 4 - true <- q4a.add(25)
22<- ! COLLISION ! -> 50 Collision Index => 1
 index => 2 - true <- q4a.add(50)
22<- ! COLLISION ! -> 43 Collision Index => 1
 index => 3 - true <- q4a.add(43)
 index => 0 - true <- q4a.add(21)
21<- ! COLLISION ! -> 7 Collision Index => 0
 index => 3 - true <- q4a.add(7)
25<- ! COLLISION ! -> 123 Collision Index => 4
 index => 3 - true <- q4a.add(123)
 index => 6 - true <- q4a.add(258)
22<- ! COLLISION ! -> 701 Collision Index => 1
 index => 1 - true <- q4a.add(701)

Is Empty: true
Size: 12
contains : true <- q4a.contains(22)
remove : true <- q4a.remove(22)
contains : false <- q4a.contains(22)
remove : false <- q4a.remove(22)
Size: 11
contains : true <- q4a.contains(135)
remove : true <- q4a.remove(135)
contains : false <- q4a.contains(135)
remove : false <- q4a.remove(135)
Size: 10
```

```
----- Size: 21 - Hash Table RecursiveHashingSet<Integer> -----
Is Empty: false

 index => 0 - true <- q4b.add(21)
 index => 1 - true <- q4b.add(22)
 index => 19 - true <- q4b.add(19)
 index => 7 - true <- q4b.add(28)
21<- ! COLLISION ! -> 63 Collision Index => 0
 index => 9 - true <- q4b.add(63)
 index => 8 - true <- q4b.add(50)
22<- ! COLLISION ! -> 43 Collision Index => 1
 index => 7 - true <- q4b.add(43)
21<- ! COLLISION ! -> 42 Collision Index => 0
 index => 6 - true <- q4b.add(42)
28<- ! COLLISION ! -> 7 Collision Index => 7
 index => 7 - true <- q4b.add(7)
 index => 18 - true <- q4b.add(123)
 index => 6 - true <- q4b.add(258)
50<- ! COLLISION ! -> 701 Collision Index => 8
 index => 17 - true <- q4b.add(701)

Is Empty: true
Size: 12
contains : true <- q4b.contains(63)
remove : true <- q4b.remove(63)
contains : false <- q4b.contains(63)
remove : false <- q4b.remove(63)
Size: 11
contains : false <- q4a.contains(135)
remove : false <- q4a.remove(135)
contains : false <- q4a.contains(135)
remove : false <- q4a.remove(135)
Size: 11

Process finished with exit code 0
```

3 Sorting Algortihms

Sort algoritmalarının kullanımında average-case testi için her bir sort algoritmasında on farklı size ve her bir size değerine ait değeri her algoritma için on kez çalıştırarak ortalamalarını aldım. Ancak worst-case durumunda 100, 1000, 5000 ve 10000 size değerlerine sahip arrayleri her bir sort algoritmasında bir kez çalıştırarak test ettim. Her algoritmanın altında çıkan sonuçlar ve grafiği bulunmaktadır.

3.1 MergeSort with DoubleLinkedList

3.1.1 Pseudocode and Explanation

3.1.1.1 MergeSortWithDoubleLinkedList. sort(LinkedList<Integer> list, int left, int right)

1. if left < right
2. Set midle to (left+right)/2.
3. Call sort(list, left, midle).
4. Call sort(list , midle + 1, right).
5. Call merge(list, left, midle, right).

sort(LinkedList<Integer> list, int left, int right)=> Parametrede alınan liste sağ ve sol listeler olarak ikiye bölünür ve recursive olarak çağrılır. Merge Sort algoritması Divide and Conquer algoritma yapısındadır. Ayrıca recursive olarak bölünerek oluşturulan listeler daha sonra Merge metodu ile birleştirilerek küçükten büyüğe doğru sıralı bir yapı oluşturur.

3.1.1.2 MergeSortWithDoubleLinkedList .merge(LinkedList<Integer> list, int left, int mid, int right)

1. Set size1 to (mid - left + 1).
2. Set size2 to (right - mid).
3. Set leftList to new LinkedList<>().
4. Set rightList to new LinkedList<>().
5. for (initialize set i to 0; condition i < size1; increment i)
6. leftList.add(list.get(left + i)).
7. for (initialize set i to 0; condition i < size2; increment i)
8. rightList.add(list.get((mid + 1+ i))).
9. Set i and j to 0.
10. Set k to left.
11. while (i < size1 and j < size2)
12. if leftList.get(i) small equal to rightList.get(j)
13. list. set(k, leftList.get(i)) and increment i.
14. else
15. list. set(k, rightList.get(j)) and increment j.

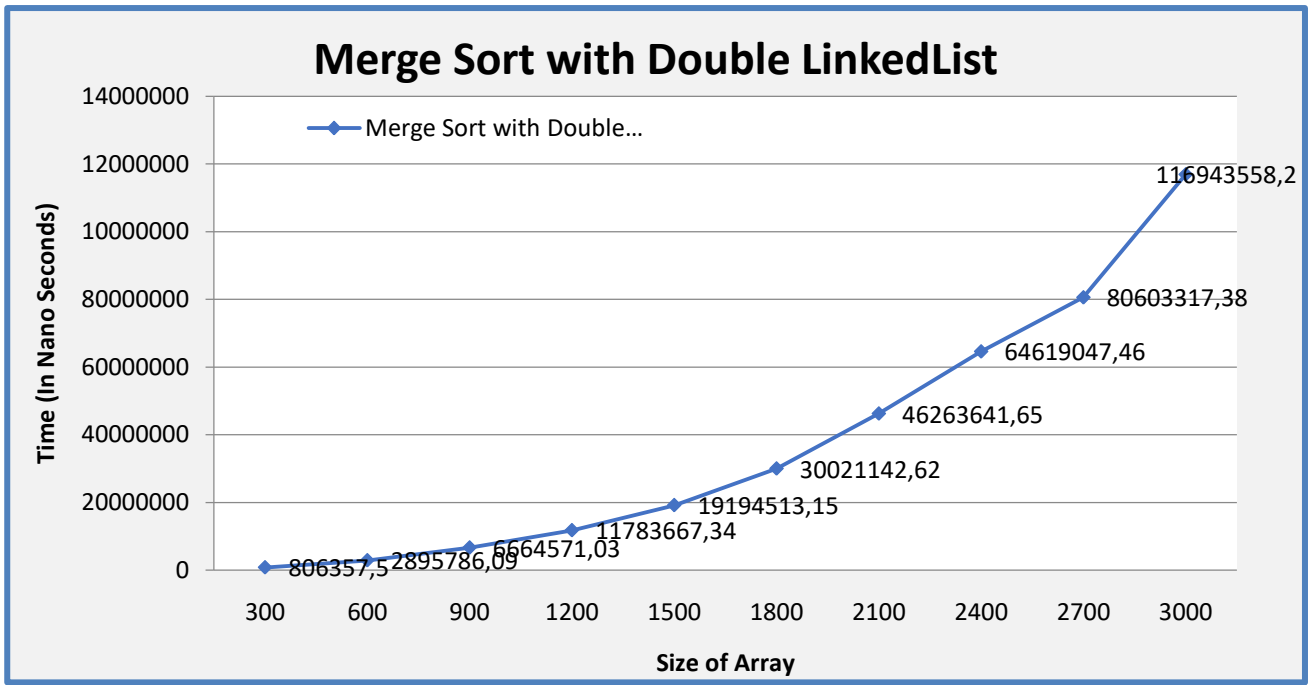
16. Increment k.
17. while (i < size1)
18. list.set(k, leftList.get(i) , increment i and k.
19. while (j < size2)
20. list.set(k, rightList.get(j)), increment j and k.

merge(LinkedList<Integer> list, int left, int mid, int right)=> Merge Sort algoritmasının temelinde listeyi bölerek işlem yaparken merge metodu ile de bu bölünmüş değerler sıralanarak birleştirilir. Son olarak sıralanmış bir liste elde edilmiş olur.

3.1.2 Average Run Time Analysis

Merge Sort with Double LinkedList		
Operation	**Size**	**AVERAGE - CASE**
1	300	Avarage of Merge Sort With Double Linked List: 806357.50
2	600	Avarage of Merge Sort With Double Linked List: 2895786.09
3	900	Avarage of Merge Sort With Double Linked List: 6664571.03
4	1200	Avarage of Merge Sort With Double Linked List: 11783667.34
5	1500	Avarage of Merge Sort With Double Linked List: 19194513.15
6	1800	Avarage of Merge Sort With Double Linked List: 30021142.62
7	2100	Avarage of Merge Sort With Double Linked List: 46263641.65
8	2400	Avarage of Merge Sort With Double Linked List: 64619047.46
9	2700	Avarage of Merge Sort With Double Linked List: 80603317.38
10	3000	Avarage of Merge Sort With Double Linked List: 116943558.23

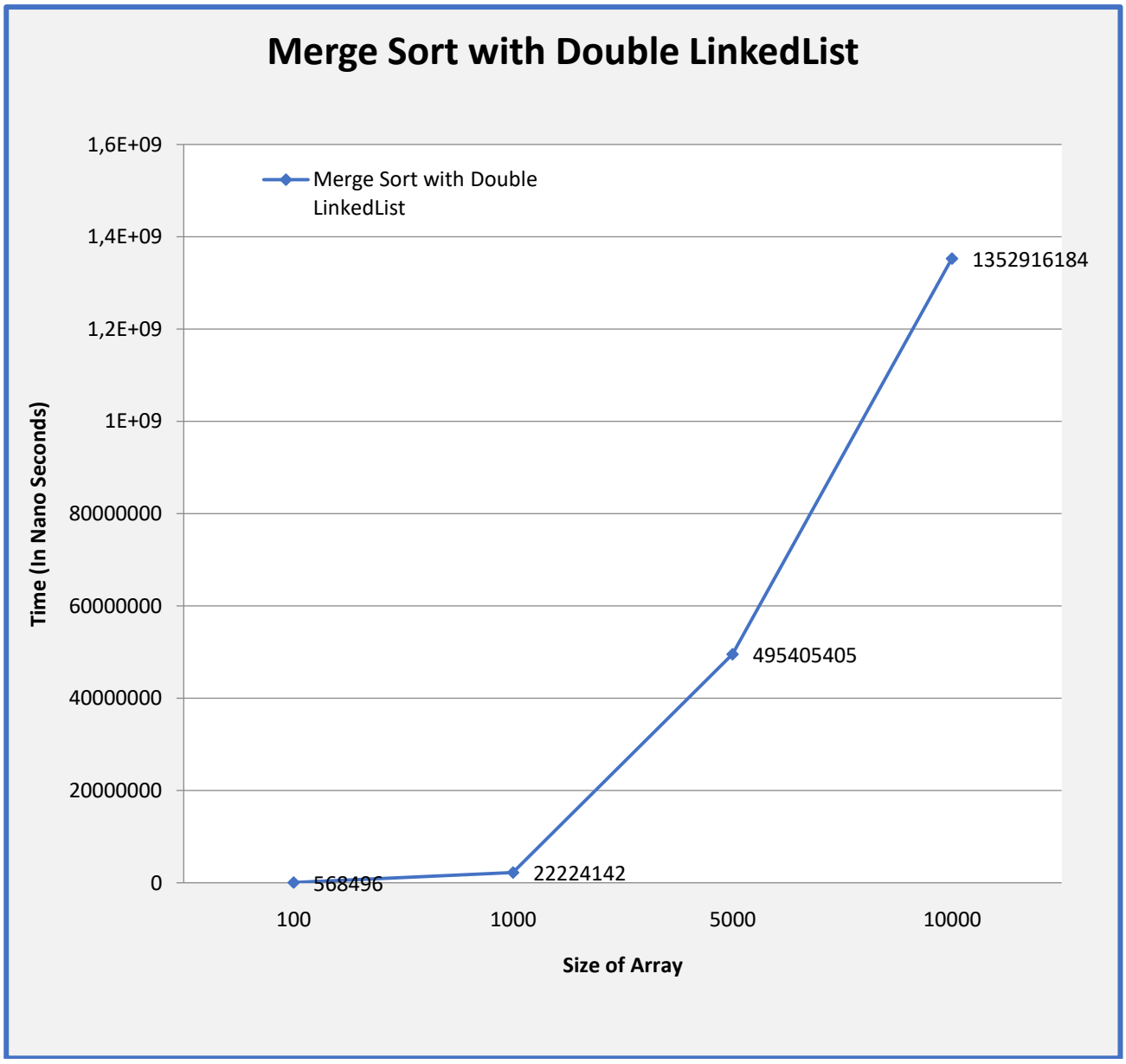
Average Run Time Graphic



3.1.3 Wort-case Performance Analysis

Merge Sort with Double LinkedList	
Size	**WORST - CASE**
100	Elapsed Nano Times: (568496)
1000	Elapsed Nano Times: (22224142)
5000	Elapsed Nano Times: (495405405)
10000	Elapsed Nano Times: (1352916184)

Worst-Case Performance Graphic

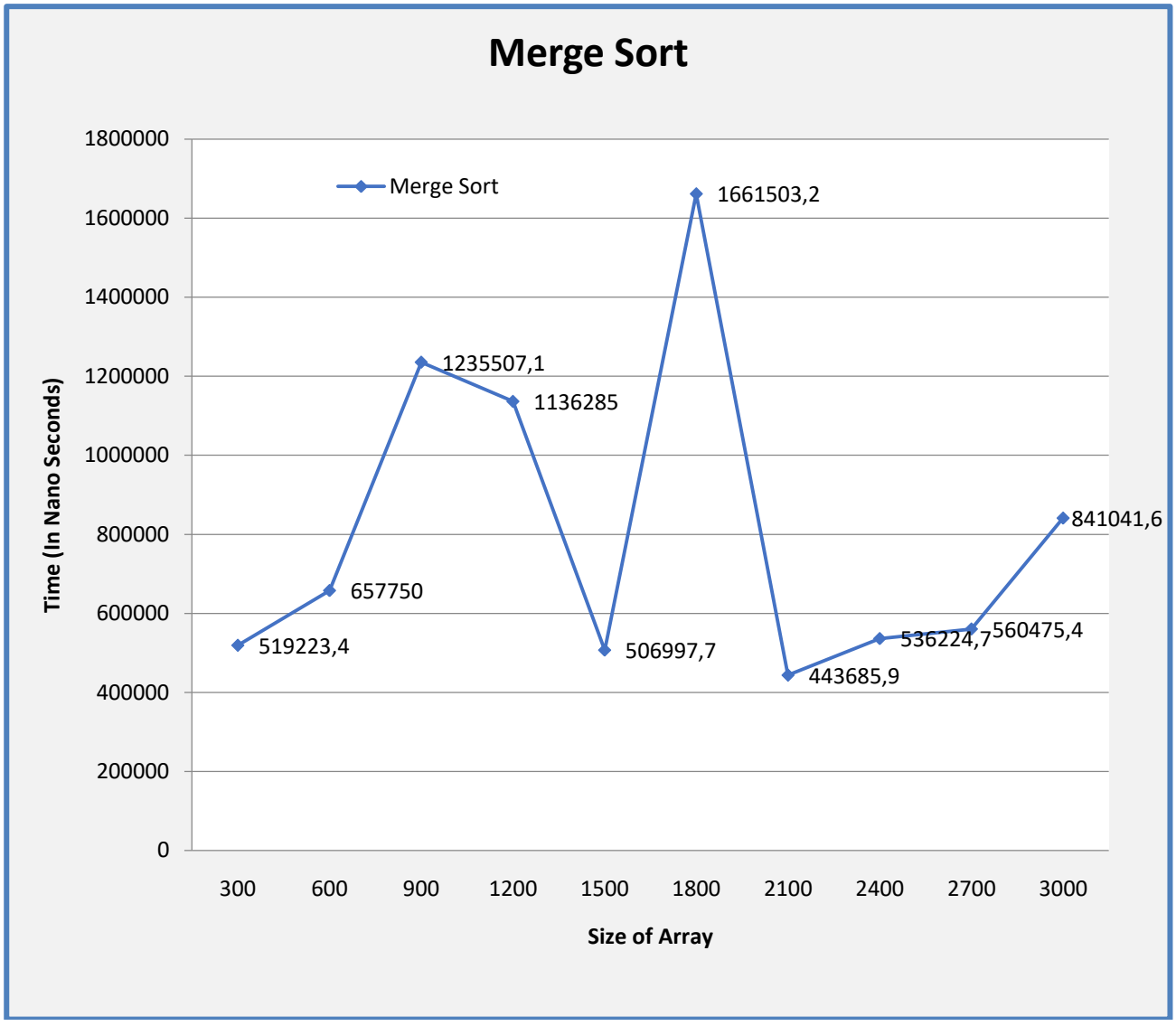


3.2 MergeSort

3.2.1 Average Run Time Analysis

Merge Sort		
Operation	**Size**	**AVERAGE - CASE**
1	300	Avarage of Merge Sort: 519223.40
2	600	Avarage of Merge Sort: 657750.00
3	900	Avarage of Merge Sort: 1235507.10
4	1200	Avarage of Merge Sort: 1136285.00
5	1500	Avarage of Merge Sort: 506997.70
6	1800	Avarage of Merge Sort: 1661503.20
7	2100	Avarage of Merge Sort: 443685.90
8	2400	Avarage of Merge Sort: 536224.70
9	2700	Avarage of Merge Sort: 560475.40
10	3000	Avarage of Merge Sort: 841041.60

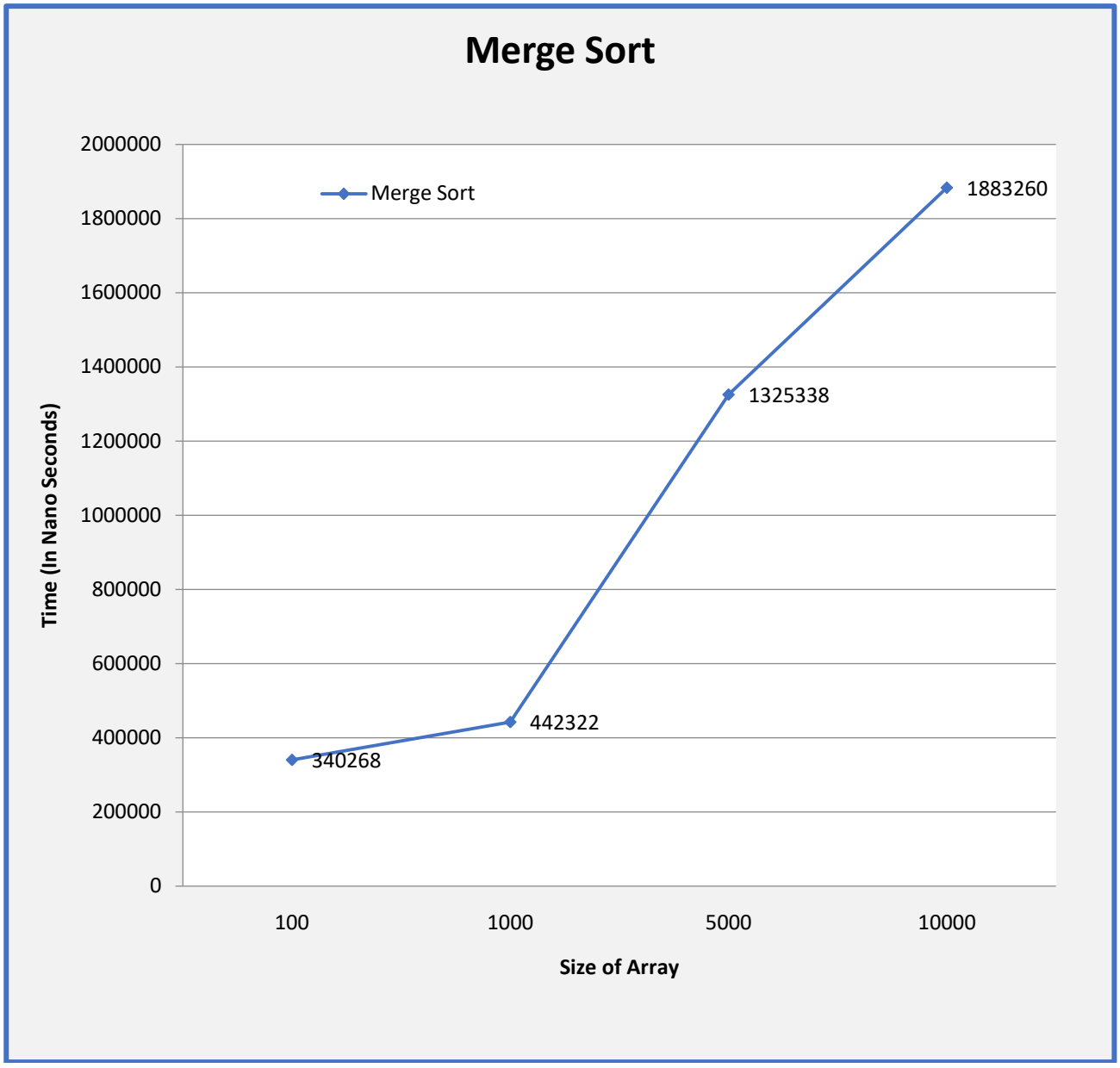
Average Run Time Graphic



3.2.2 Wort-case Performance Analysis

Merge Sort	
Size	**WORST - CASE**
100	Elapsed Nano Times: (340268)
1000	Elapsed Nano Times: (442322)
5000	Elapsed Nano Times: (1325338)
10000	Elapsed Nano Times: (1883260)

Worst-Case Performance Graphic

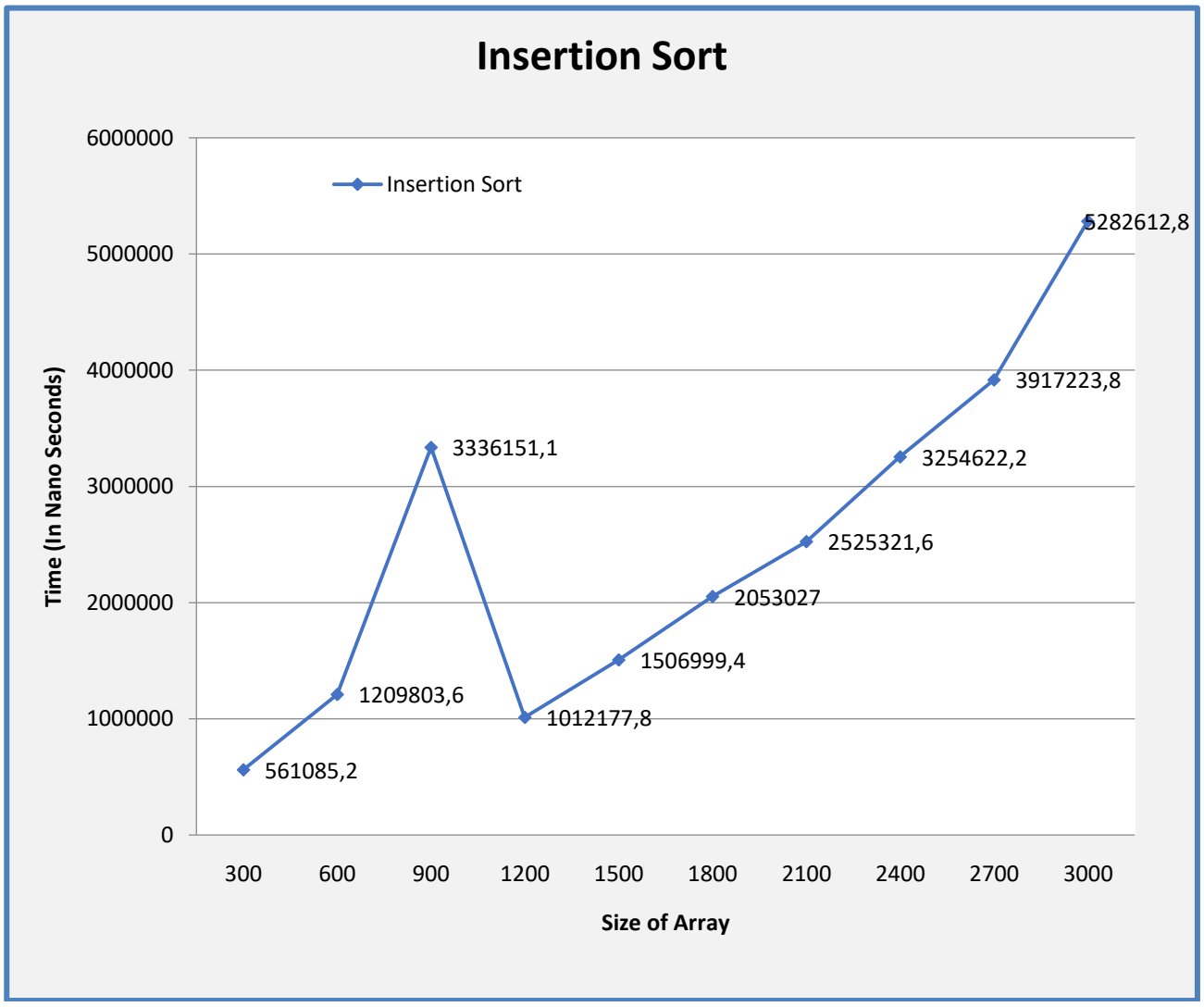


3.3 Insertion Sort

3.3.1 Average Run Time Analysis

Insertion Sort		
Operation	**Size**	**AVERAGE - CASE**
1	300	Avarage of Insertion Sort: 561085.20
2	600	Avarage of Insertion Sort: 1209803.60
3	900	Avarage of Insertion Sort: 3336151.10
4	1200	Avarage of Insertion Sort: 1012177.80
5	1500	Avarage of Insertion Sort: 1506999.40
6	1800	Avarage of Insertion Sort: 2053027.00
7	2100	Avarage of Insertion Sort: 2525321.60
8	2400	Avarage of Insertion Sort: 3254622.20
9	2700	Avarage of Insertion Sort: 3917223.80
10	3000	Avarage of Insertion Sort: 5282612.80

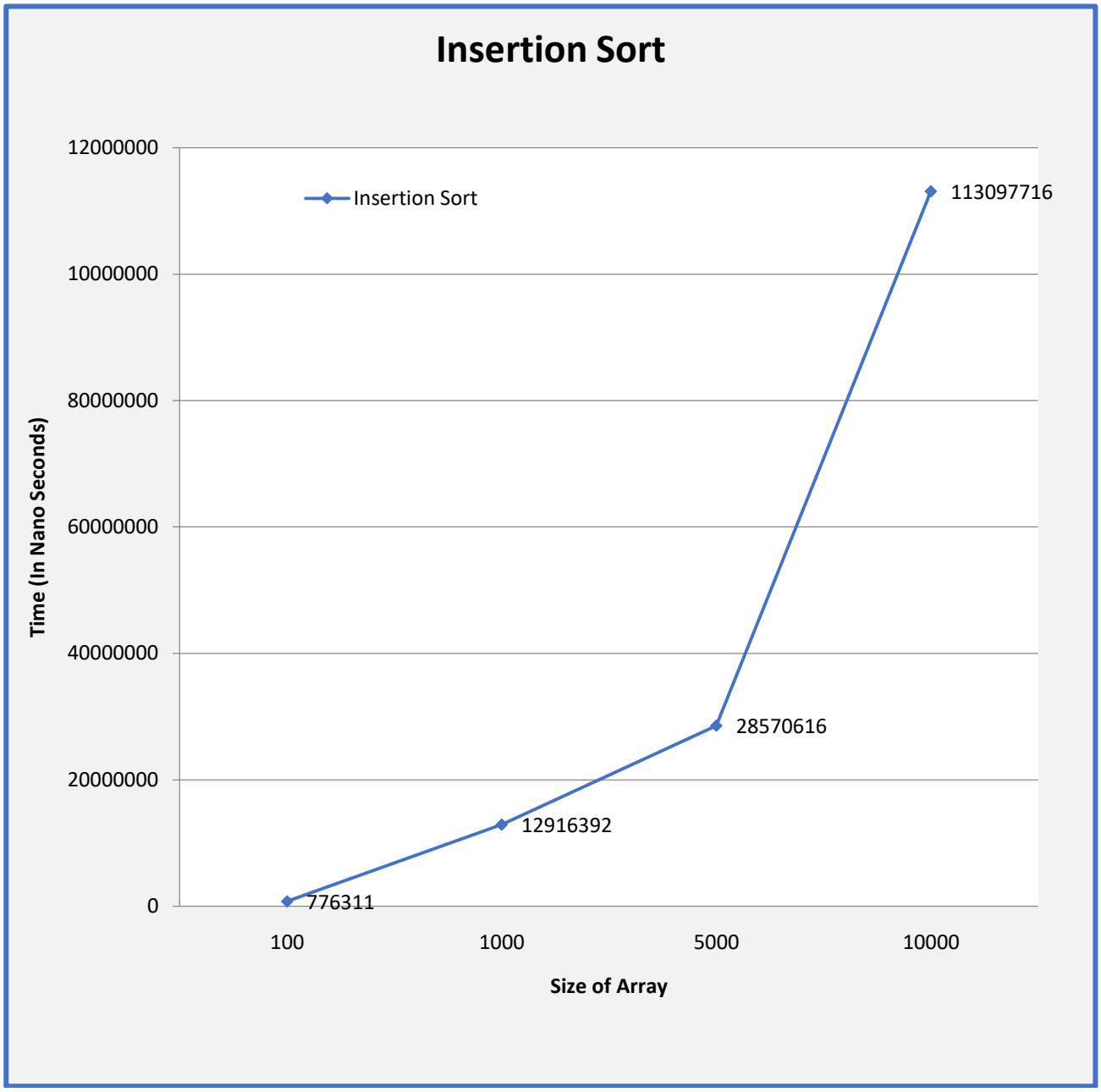
Average Run Time Graphic



3.3.2 Wort-case Performance Analysis

Insertion Sort	
Size	**WORST - CASE**
100	Elapsed Nano Times: (776311)
1000	Elapsed Nano Times: (12916392)
5000	Elapsed Nano Times: (28570616)
10000	Elapsed Nano Times: (113097716)

Worst-Case Performance Graphic

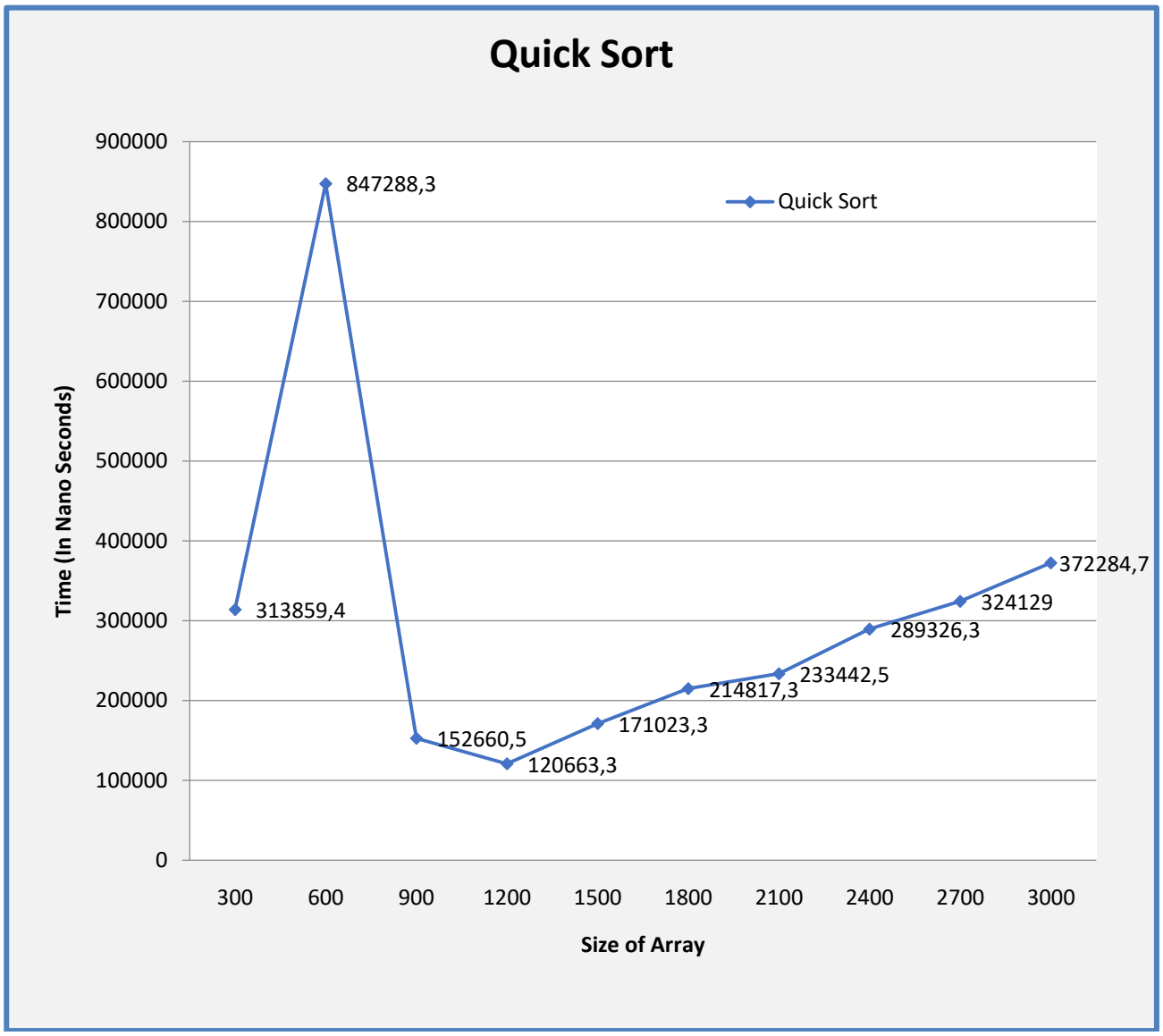


3.4 Quick Sort

3.4.1 Average Run Time Analysis

Quick Sort		
Operation	**Size**	**AVERAGE - CASE**
1	300	Avarage of Quick Sort: 313859.40
2	600	Avarage of Quick Sort: 847288.30
3	900	Avarage of Quick Sort: 152660.50
4	1200	Avarage of Quick Sort: 120663.30
5	1500	Avarage of Quick Sort: 171023.30
6	1800	Avarage of Quick Sort: 214817.30
7	2100	Avarage of Quick Sort: 233442.50
8	2400	Avarage of Quick Sort: 289326.30
9	2700	Avarage of Quick Sort: 324129.00
10	3000	Avarage of Quick Sort: 372284.70

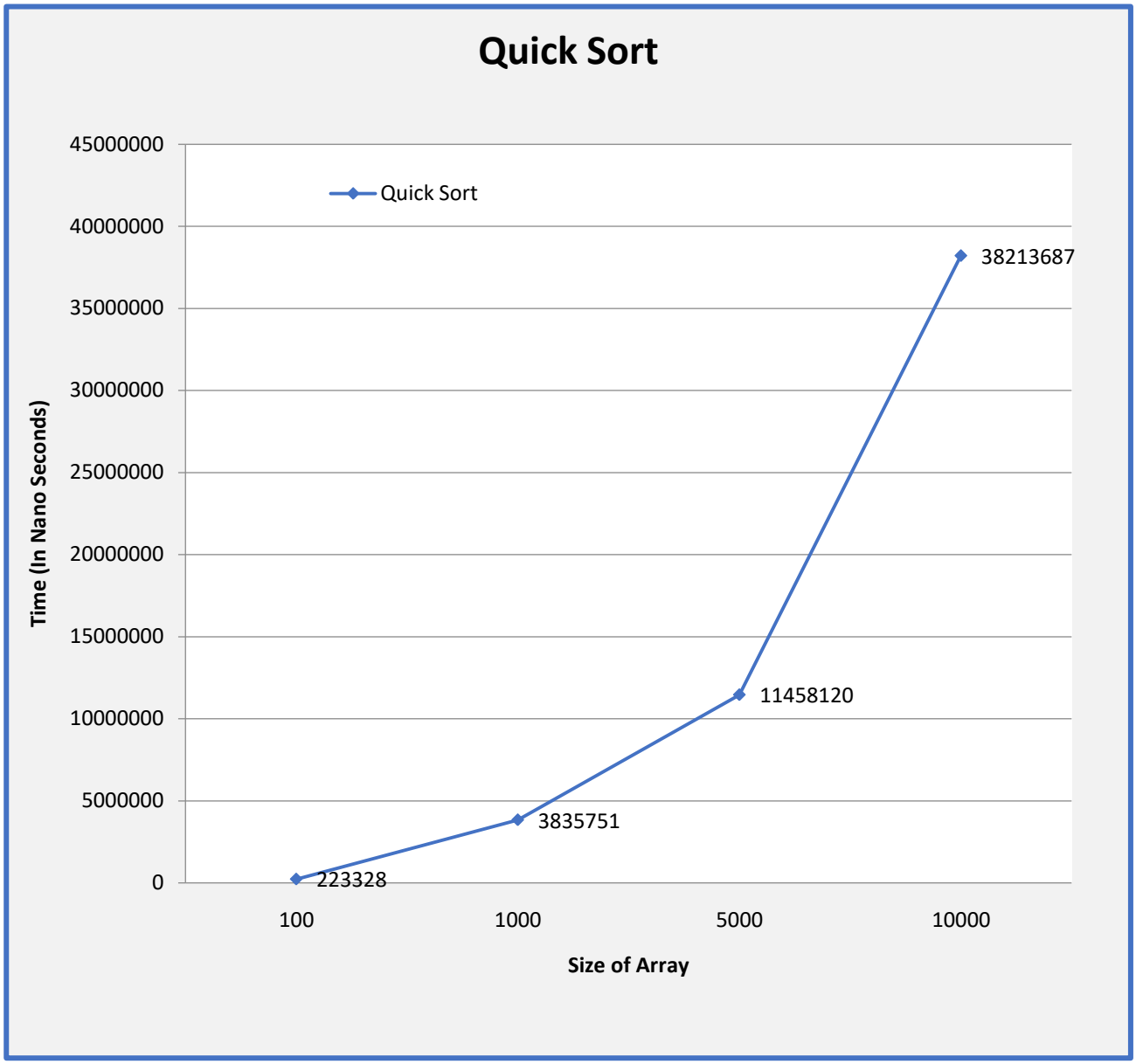
Average Run Time Graphic



3.4.2 Wort-case Performance Analysis

Quick Sort	
Size	**WORST - CASE**
100	Elapsed Nano Times: (223328)
1000	Elapsed Nano Times: (3835751)
5000	Elapsed Nano Times: (11458120)
10000	Elapsed Nano Times: (38213687)

Worst-Case Performance Graphic

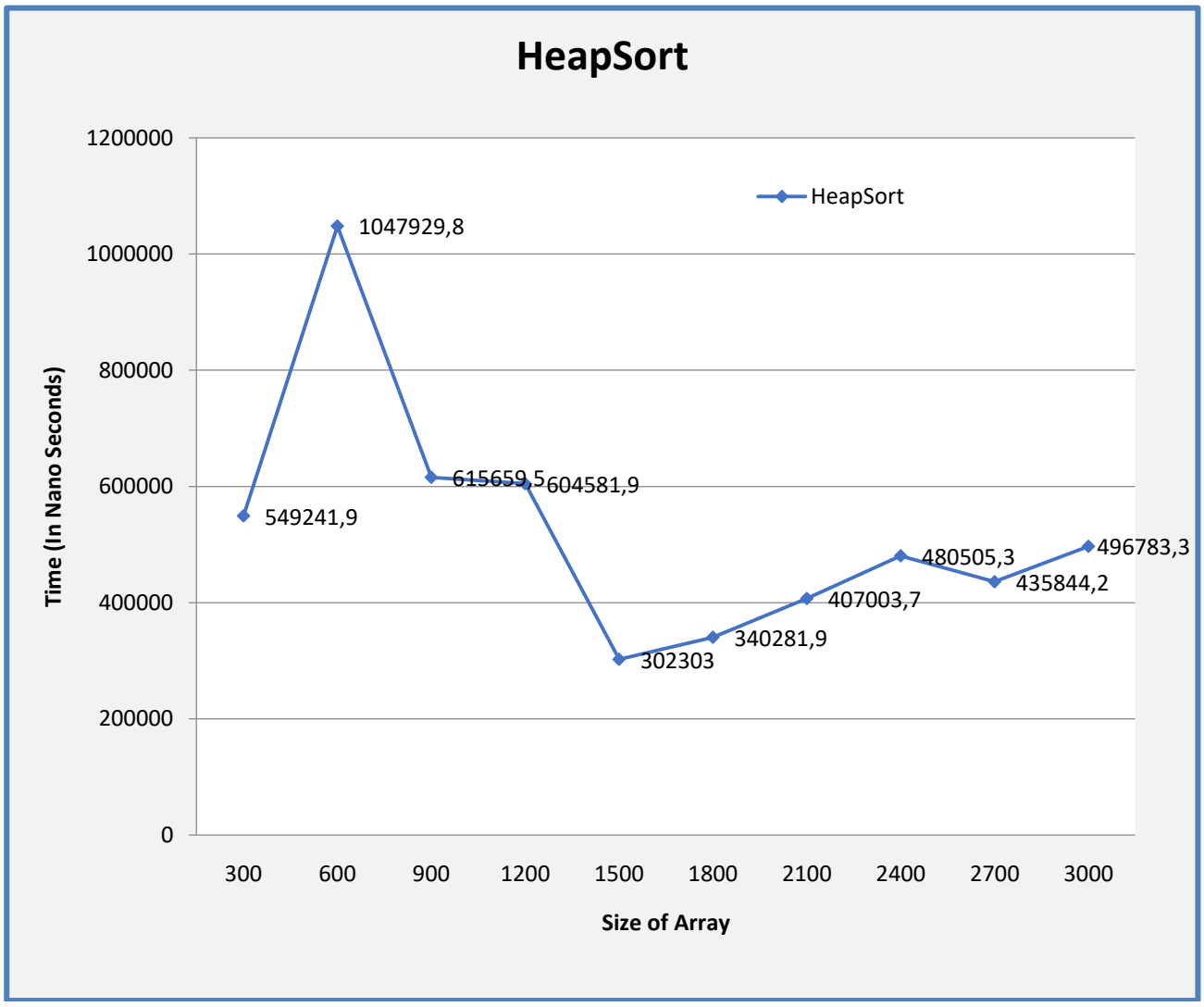


3.5 Heap Sort

3.5.1 Average Run Time Analysis

Heap Sort		
Operation	**Size**	**AVERAGE - CASE**
1	300	Avarage of Heap Sort: 549241.90
2	600	Avarage of Heap Sort: 1047929.80
3	900	Avarage of Heap Sort: 615659.50
4	1200	Avarage of Heap Sort: 604581.90
5	1500	Avarage of Heap Sort: 302303.00
6	1800	Avarage of Heap Sort: 340281.90
7	2100	Avarage of Heap Sort: 407003.70
8	2400	Avarage of Heap Sort: 480505.30
9	2700	Avarage of Heap Sort: 435844.20
10	3000	Avarage of Heap Sort: 496783.30

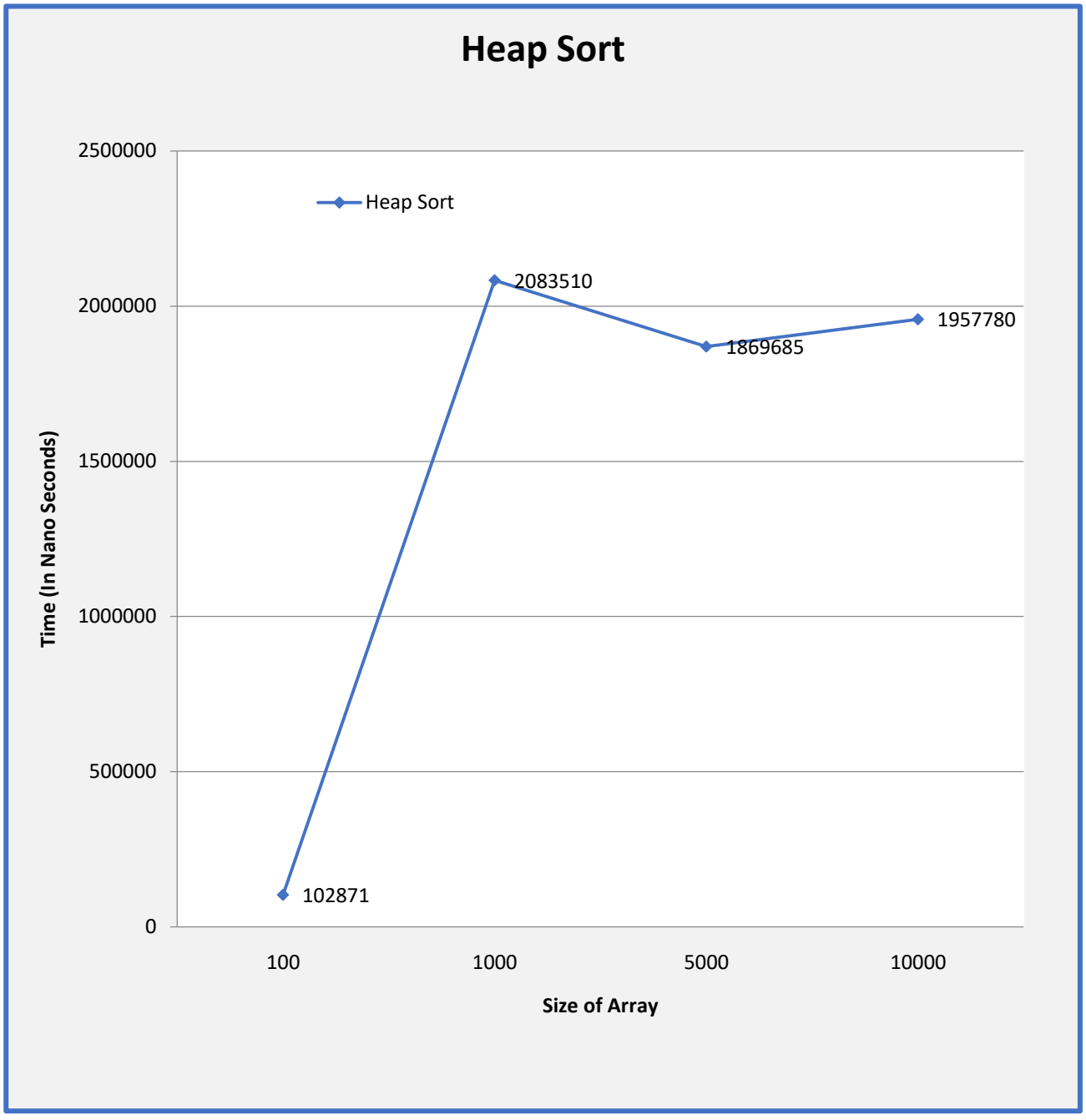
Average-Case Run Time Graphic



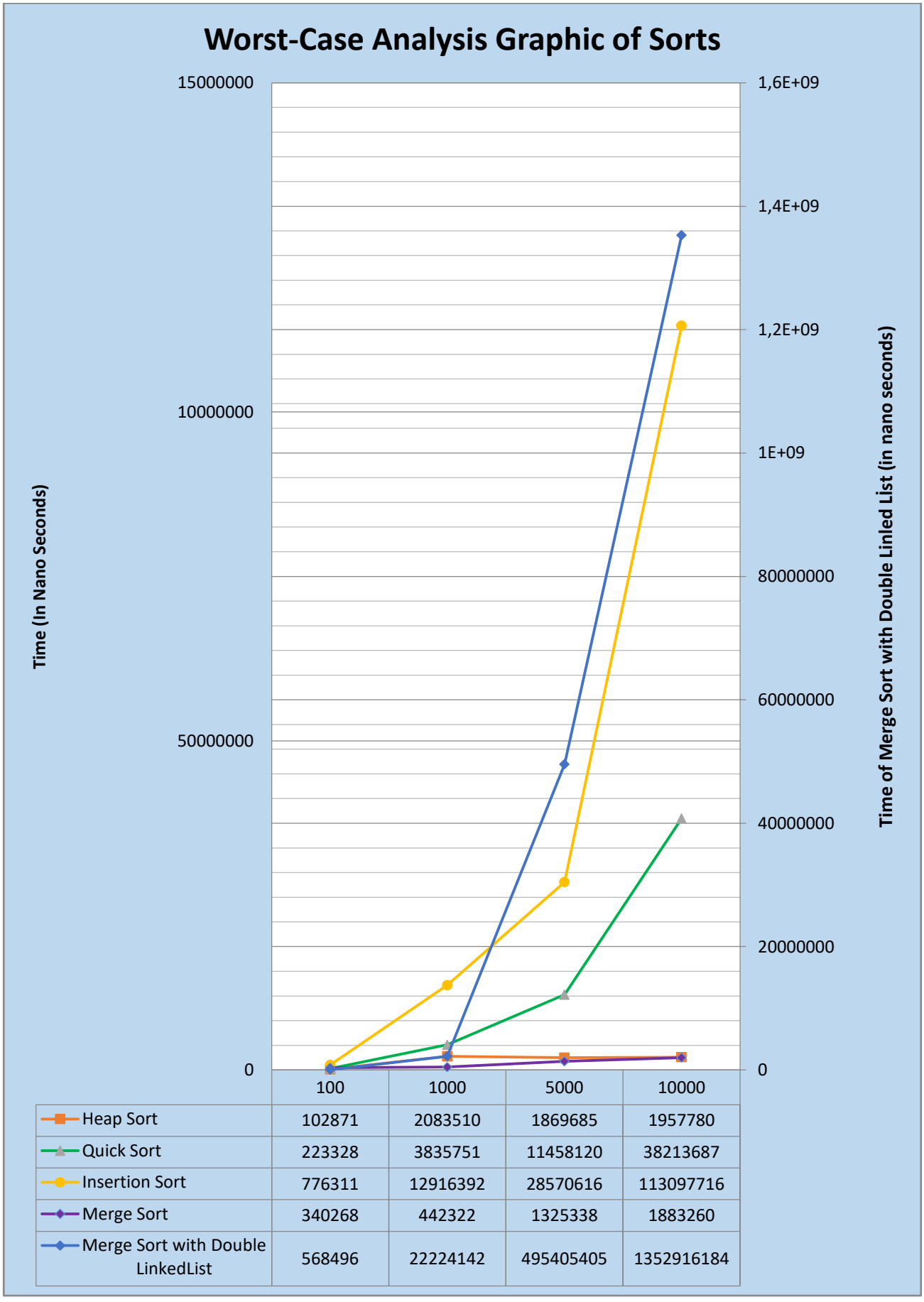
3.5.2 Wort-case Performance Analysis

Heap Sort	
Size	**WORST - CASE**
100	Elapsed Nano Times: (102871)
1000	Elapsed Nano Times: (2083510)
5000	Elapsed Nano Times: (1869685)
10000	Elapsed Nano Times: (1957780)

Worst-Case Performance Graphic



4 Comparison the Analysis Results



	100	1000	5000	10000
Heap Sort	102871	2083510	1869685	1957780
Quick Sort	223328	3835751	11458120	38213687
Insertion Sort	776311	12916392	28570616	113097716
Merge Sort	340268	442322	1325338	1883260
Merge Sort with Double LinkedList	568496	22224142	495405405	1352916184

Grafik Sonucu => Yukarıda belirtilen grafikte tüm algoritmaların nano saniye cinsinden ölçümlerinden çıkan sonuca göre Merge Sort Double LinkedList logaritmik olarak worst – case durumunda size 1000' e kadar Insertion Sort ve Quick Sort' a göre iyi çalışmasına rağmen size 1000 ile 10000 arasında Insertion Sort' tan bile daha kötü bir sıralama yapmaktadır. Grafiğe göre worst – case değerlerine göre Merge Sort ve Heap Sort iyi bir sıralama grafiği çizmişlerdir. Ancak bu grafikte nano saniye değerlerinden dolayı tam olarak belirli olmasa da Section 3 – Sorting Analysis kısmında tüm değerler ve grafikler de mevcuttur. O kısımda daha net bir analysis sonucuna ulaşılır. Ayrıca bu nano saniye değerler CPU gücüne bağlı olarak da değişebilir. O esnada sort işlemi yapılırken arka planda yapılan başka bir işlemden dolayı değerlerde ufak bir sapma olabilir.

Not: Tüm partların Screen Shot' ları ayrıca bir klasörde yüklenmiştir.