

UNIVERSITY OF NATURAL SICIENCE

REPORT

Sorting Algorithms

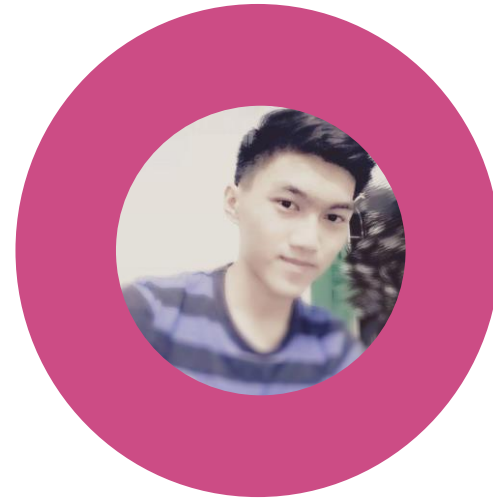
Sorting Algorithms Overview

1. Selection sort
2. Insertion sort
3. Bubble sort
4. Quick sort
5. Merge sort
6. Heap sort
7. Radix sort
8. Counting sort
9. Shell sort
10. Shaker sort
11. Flash sort
12. Binary Insertion sort

ID: 1753025



I'm
Quoc An





SELECTION SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Choose a smallest in N elements of an array, bring it to the first position of an array.

After that, we don't care about this element anymore. Let see this array only have N-1 elements and start from the second position.

Repeat this process until this array has one element left.

Let summarize it following these steps.

Step 1: let $i = 1$;

Step 2: Find the smallest element $a[\text{min}]$ in the current array from i to n .

Step 3: Change position of $a[\text{min}]$ and $a[i]$

Step 4: If $i < n$ then $i = i + 1$. Repeat step 2
Otherwise: Stop ($n - 1$ elements place on right position)

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int i, j, min_pos;

for (i = 0; i < n - 1; i++) {
    min_pos = i;

    for (j = i + 1; j < n; j++) {
        if (arr[j] < arr[min_pos]) {
            min_pos = j;
        }
    }

    swap(&arr[min_pos], &arr[i]);
}
```

ALGORITHM REVIEW

In Selection sort algorithm, we can recognize that in every i , we will need $(n - i)$ times to determine the smallest element. The number of comparisons do not depend on the status of the array, thus in every case we will have:

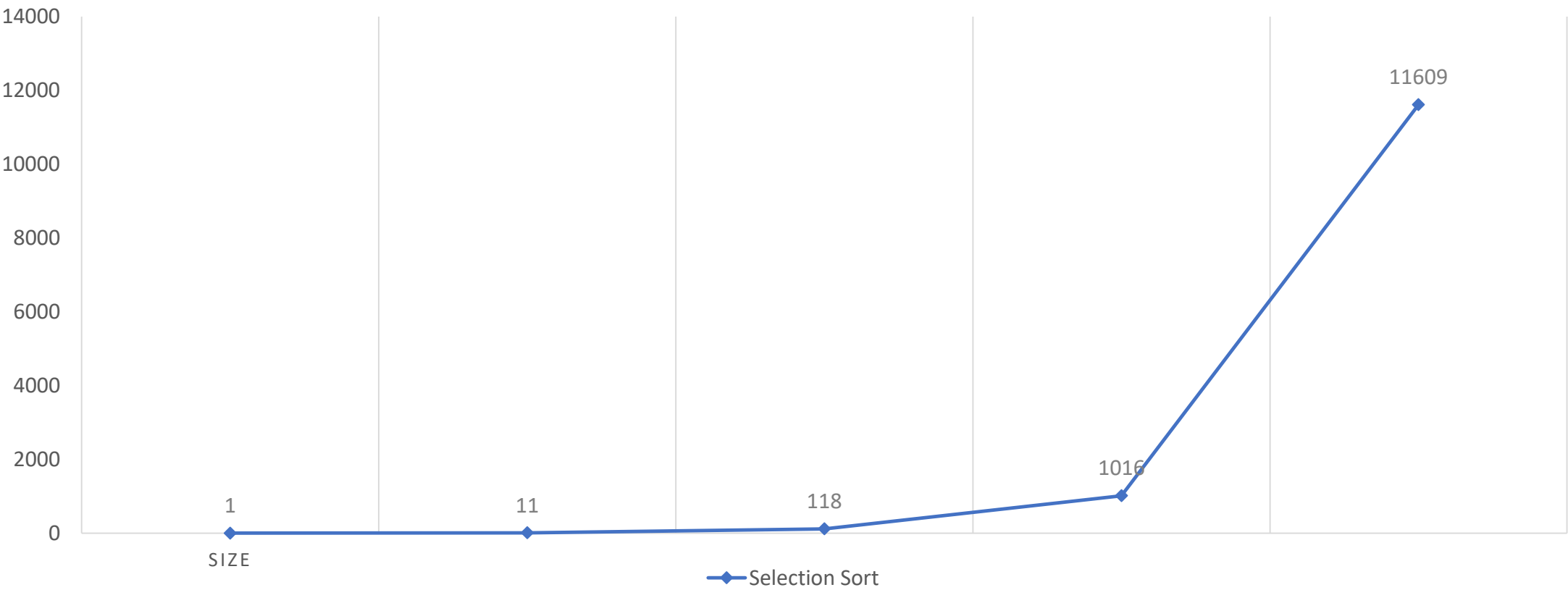
Comparison times: $\frac{n(n-1)}{2}$

Case	Time complexity
Best	$O(n^2)$
Worst	$O(n^2)$

It is nested two loop

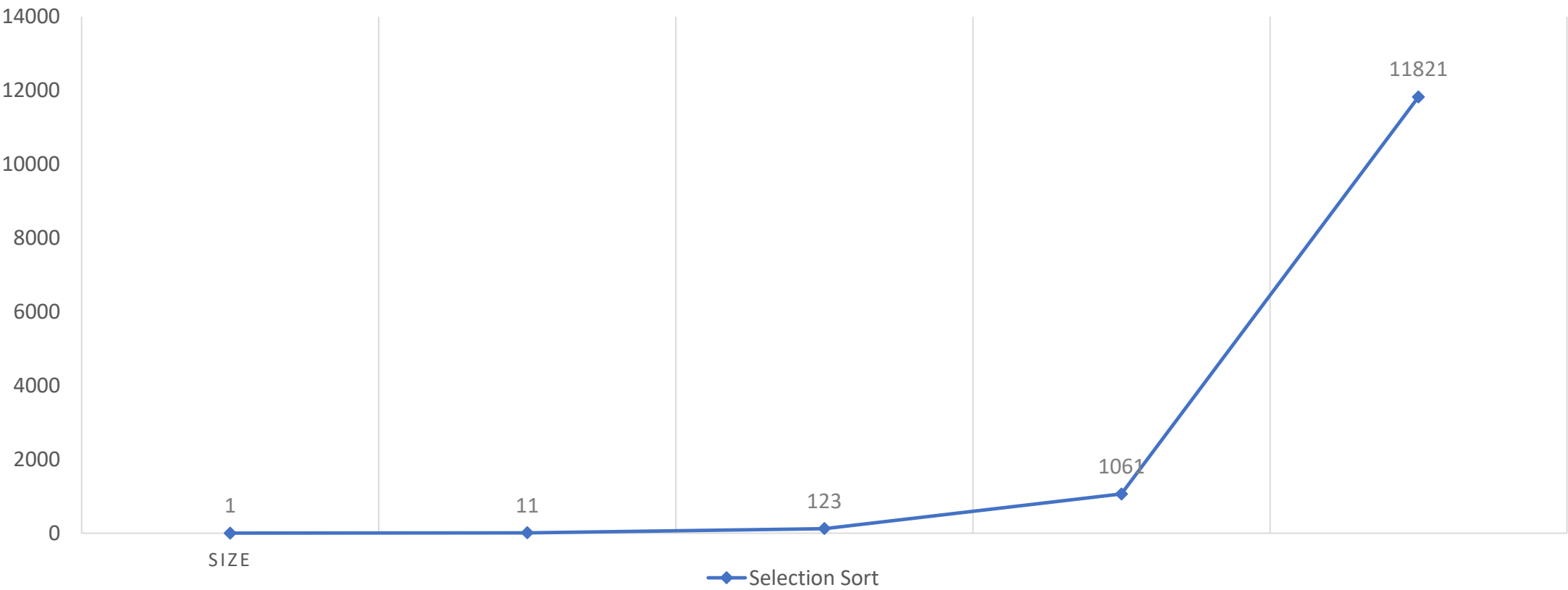
GRAPH

RANDOM DATA TYPE



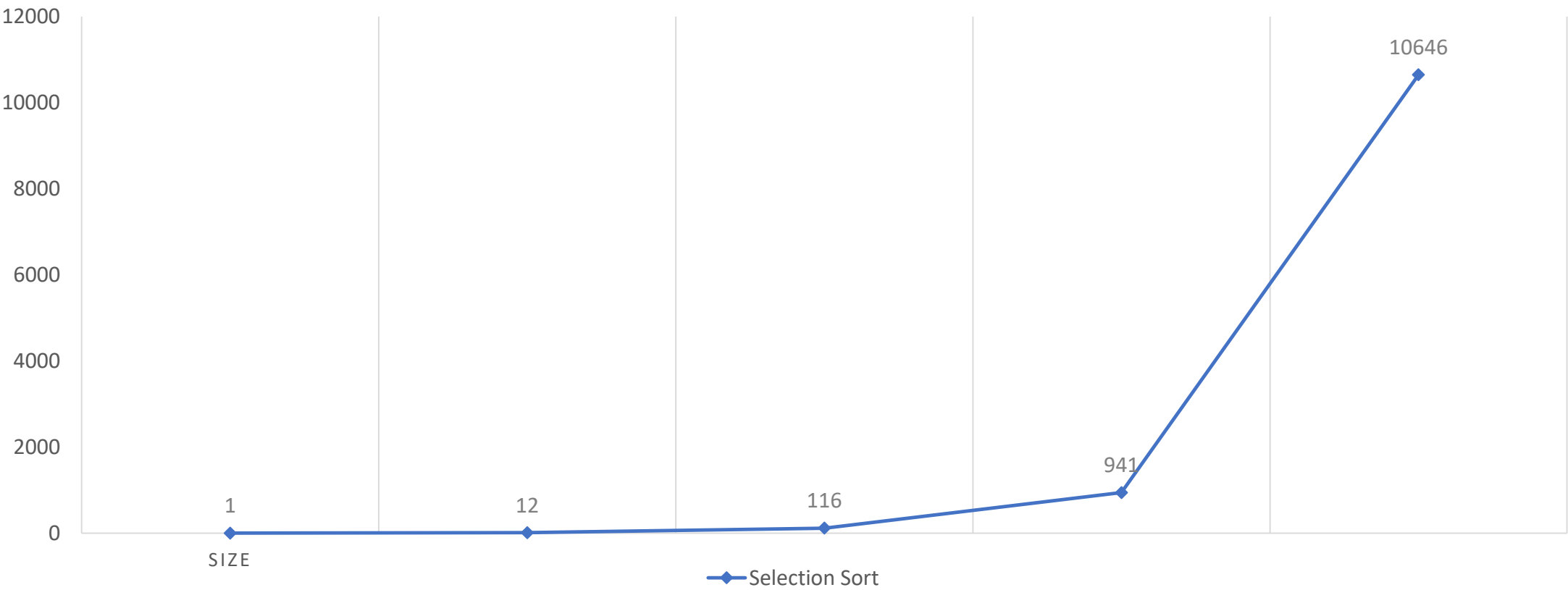
GRAPH

SORTED DATA TYPE



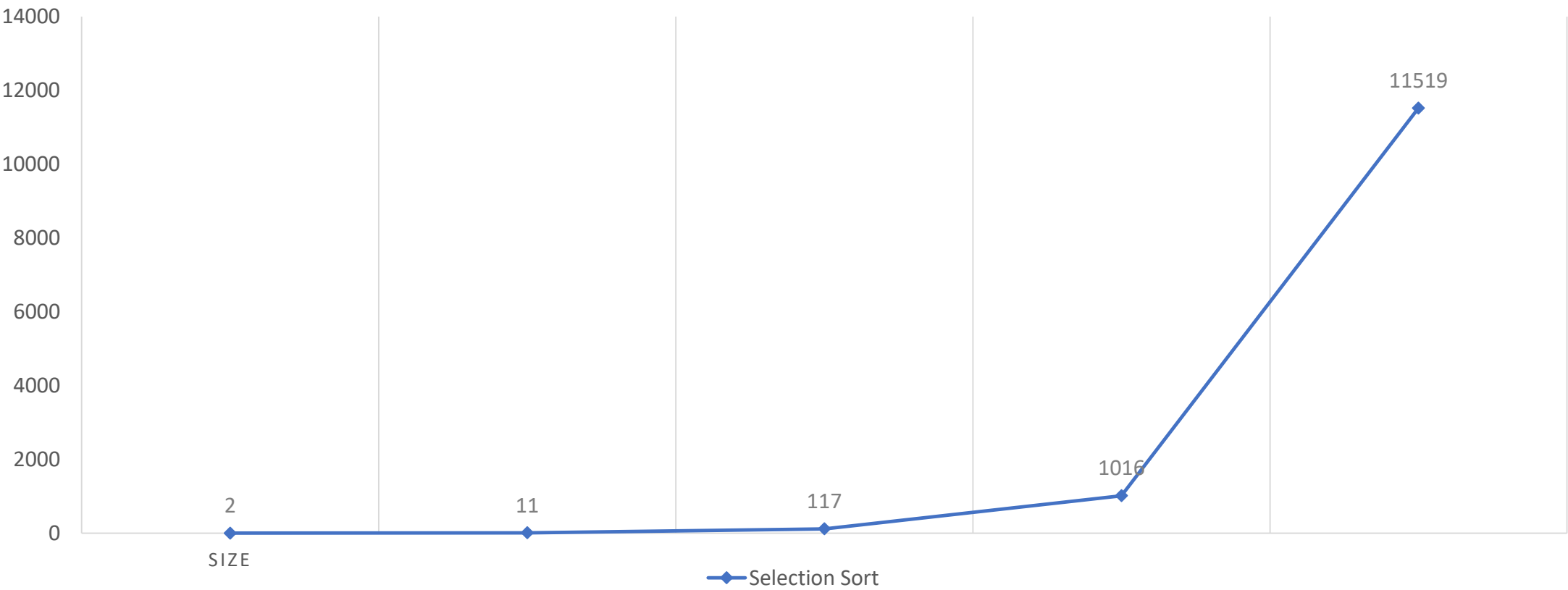
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





INSERTION SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

IDEA

Find the way to insert $a[i]$ into suitable position in sorted part to have a new array a_1, a_2, \dots, a_n is orderly.. This is the position between 2 elements a_{m-1} and a_m with $a_{m-1} < a_i < a_m$ ($1 \leq m \leq i$)

Given an array with n elements, we can know that this array already have a_1 is sorted, then we add a_2 into the sorted part a_1 , then we will have a_1, a_2 is sorted. Continue doing this until add the element an into sorted part a_1, a_2, \dots, a_{n-1} .

At last, we will have the sorted array with n elements.

Step 1: let $i = 2$; //suppose that we already had part $a[1]$ is sorted.

Step 2: $x = a[i]$, Find the position in sorted part $a[1]$ to $a[i-1]$ to insert $a[i]$ into it.

Step 3: Displace all elements from $a[\text{pos}]$ to $a[i-1]$ to right 1 unit to take place for $a[i]$ get in.

Step 4: $a[\text{pos}] = x$

Step 5: $i = i + 1$;
If $i \leq n$: Repeat step 2
Else: Stop

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int j, key;

for (int i = 0; i < n; i++) {
    key = arr[i];
    j = i - 1;

    while ((j >= 0) && (arr[j] > key)) {
        arr[j + 1] = arr[j];
        j -= 1;
    }
    arr[j + 1] = key;
}
```

ALGORITHM REVIEW

In Insertion sort algorithm, the comparisons appear in every while loop to find the position and every time determine the position, if the position is checking not suitable, we will displace element $a[pos]$.

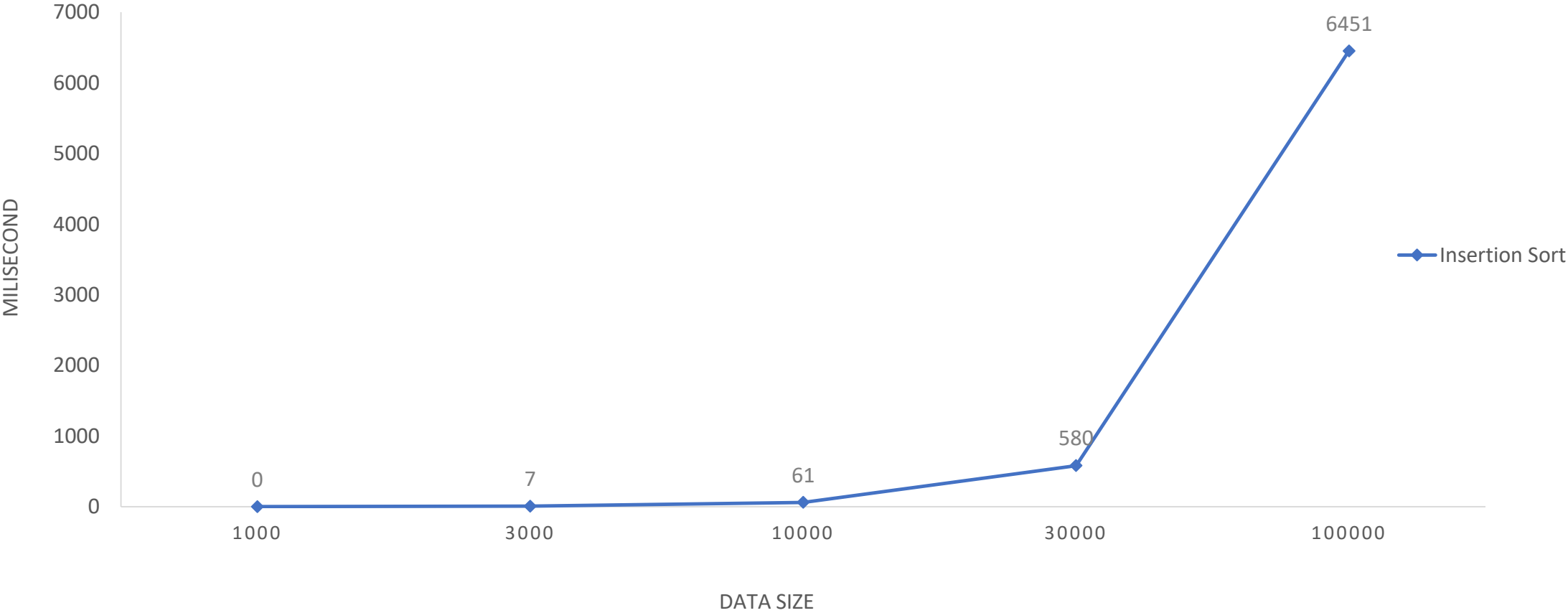
This algorithm will do $n-1$ while loop and the number of comparison depend on the status of the array (random, sorted, reverse or nearly sorted data).

Best case of insertion sort is $O(n)$ when array is already sorted

Case	Time complexity
Best	$O(n)$
Worst	$O(n^2)$
Average	$O(n^2)$

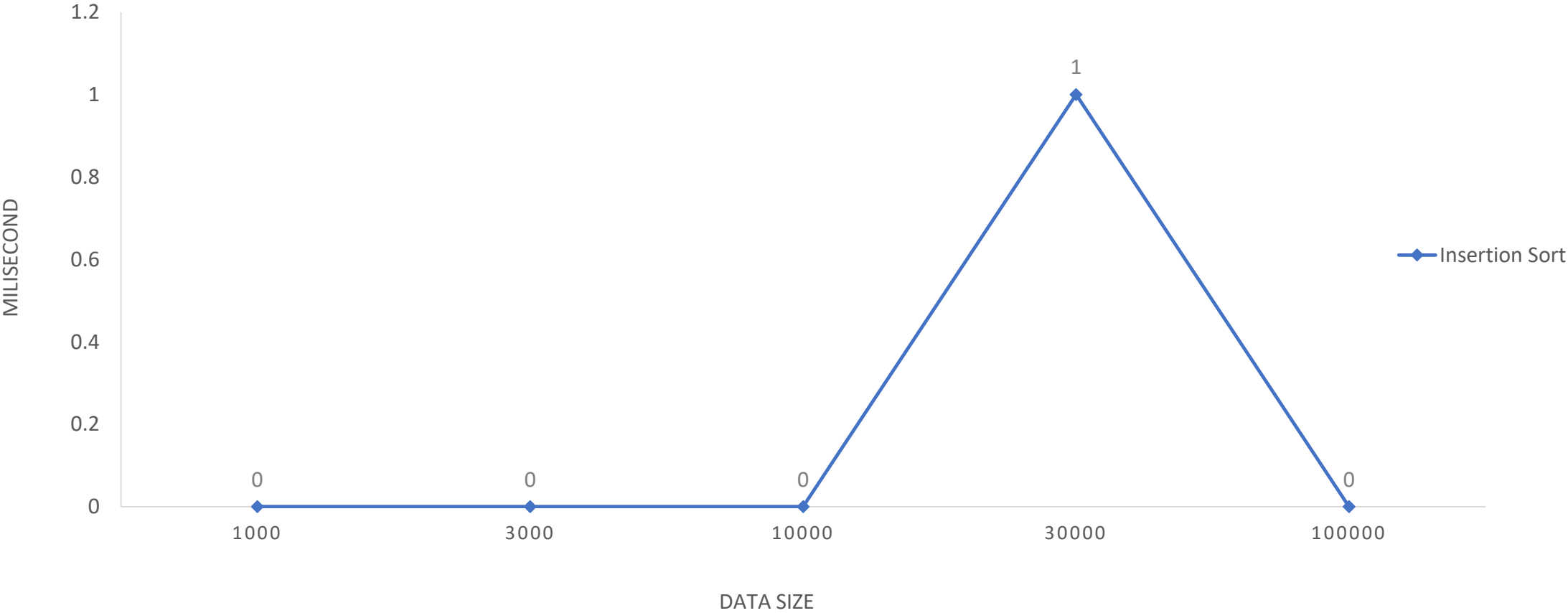
GRAPH

RANDOM DATA TYPE



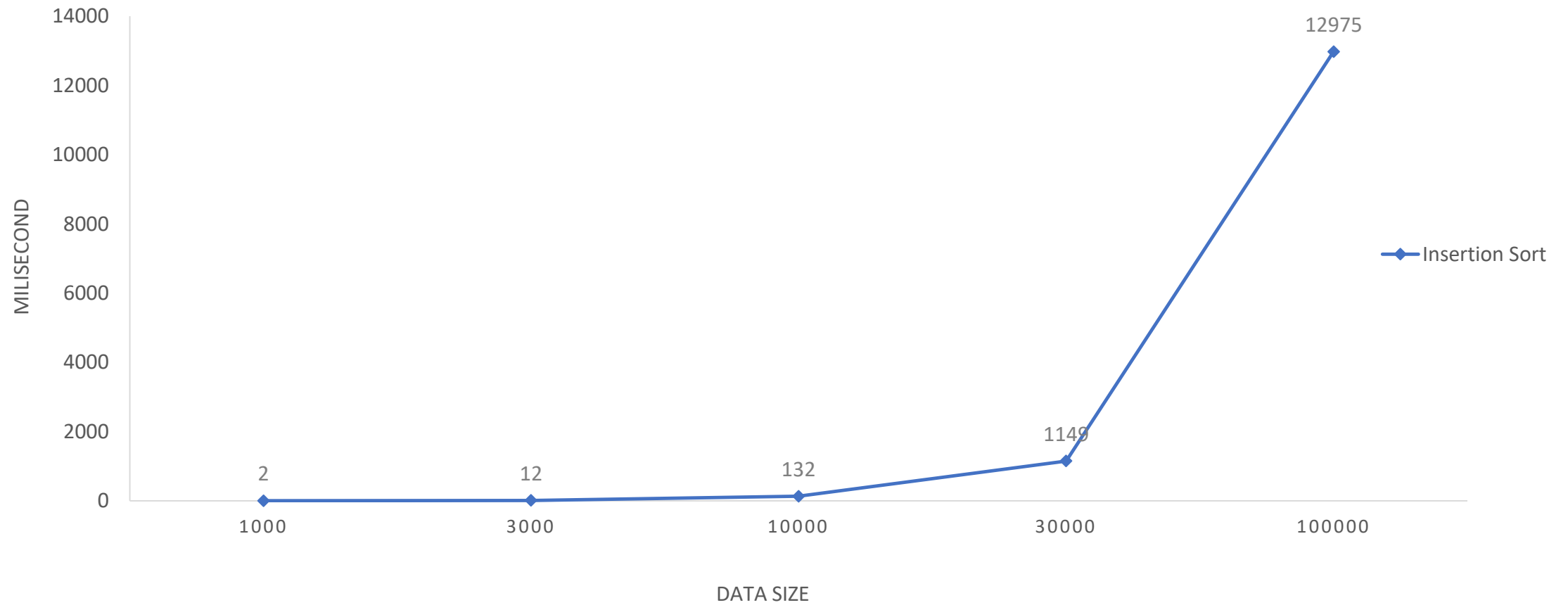
GRAPH

SORTED DATA TYPE



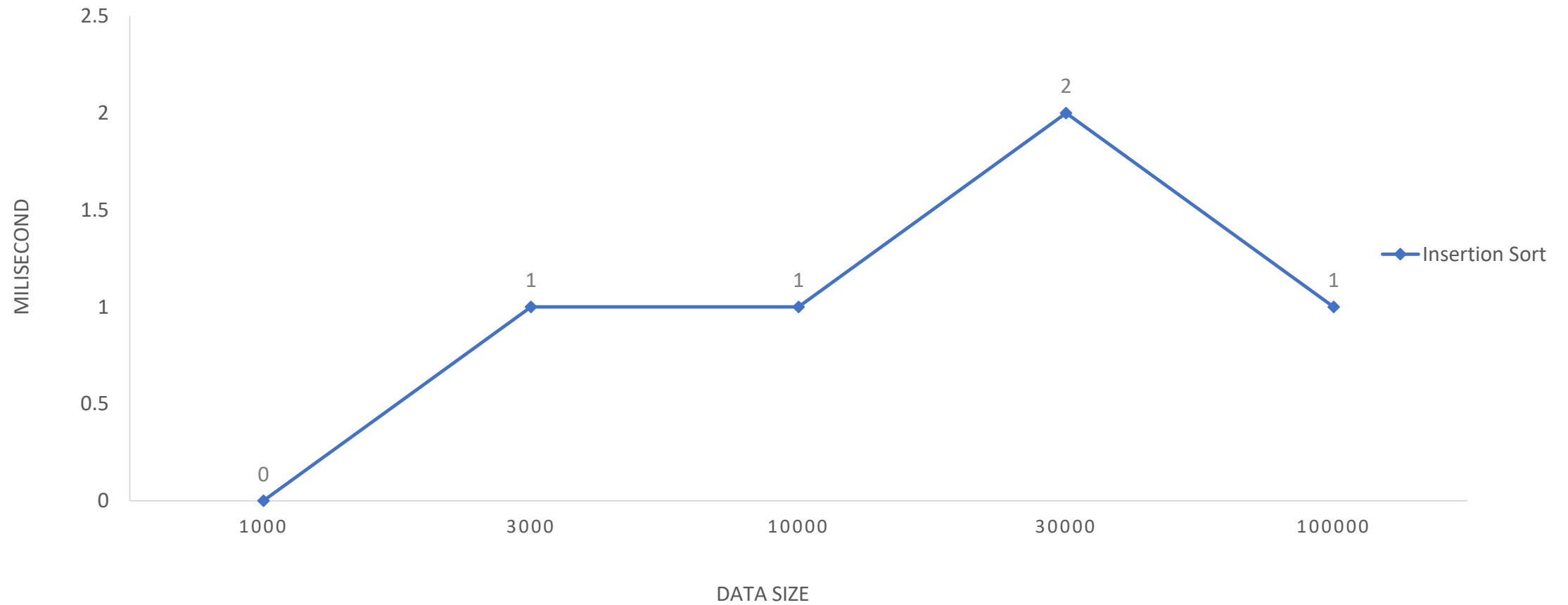
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





BUBBLE SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Start from the begin or the end of the array, change the position of pair elements to bring the smaller (bigger) element to the beginning of the array or the end of the array.

Continue doing this until we have a sorted array.

Step 1: let $i = 0$;

Step 2: $j = i$;
When $j < n - 1$
if $a[j] > a[j + 1]$: swap 2 elements
 $j += 1$;

Step 3: $i++$;
If $i > n - 1$: Stop
else do Step 2

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
bool isSwap = false;

for (int i = 0; i < n - 1; i++) {
    isSwap = false;

    for (int j = i; j < n-1; j++) {
        if (arr[j] > arr[j + 1]) {
            swap(&arr[j], &arr[j + 1]);
            isSwap = true;
        }
    }
    if (isSwap == false)
        break;
}
```

ALGORITHM REVIEW

In bubble sort algorithm, the number of comparisons do not depend on the status of the array.

Disadvantage: Can not recognize the sorted array

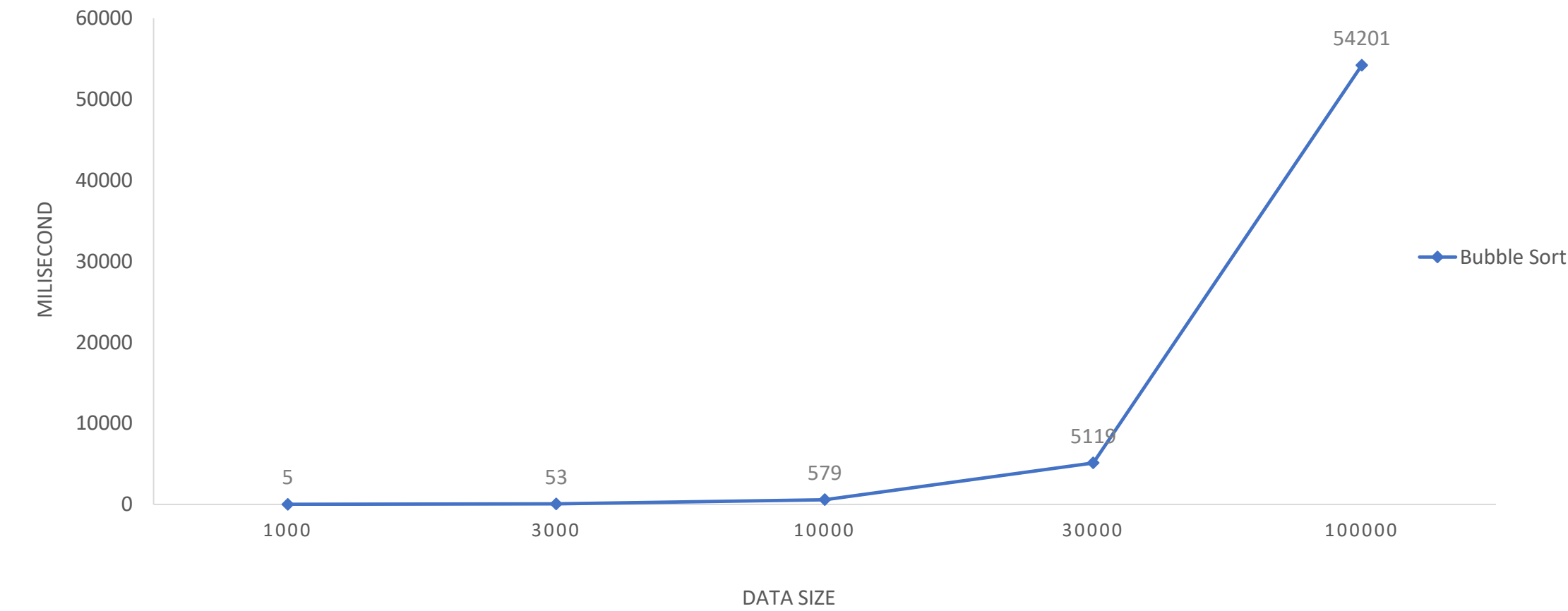
But we can add one variable: *bool isSwap = false*, if we don't do any swap in the first run, we will break because the array is already sorted.

So, we can reduce the running time in this case.

Case	Time complexity
Best	$O(n)$
Worst	$O(n^2)$
Average	$O(n^2)$

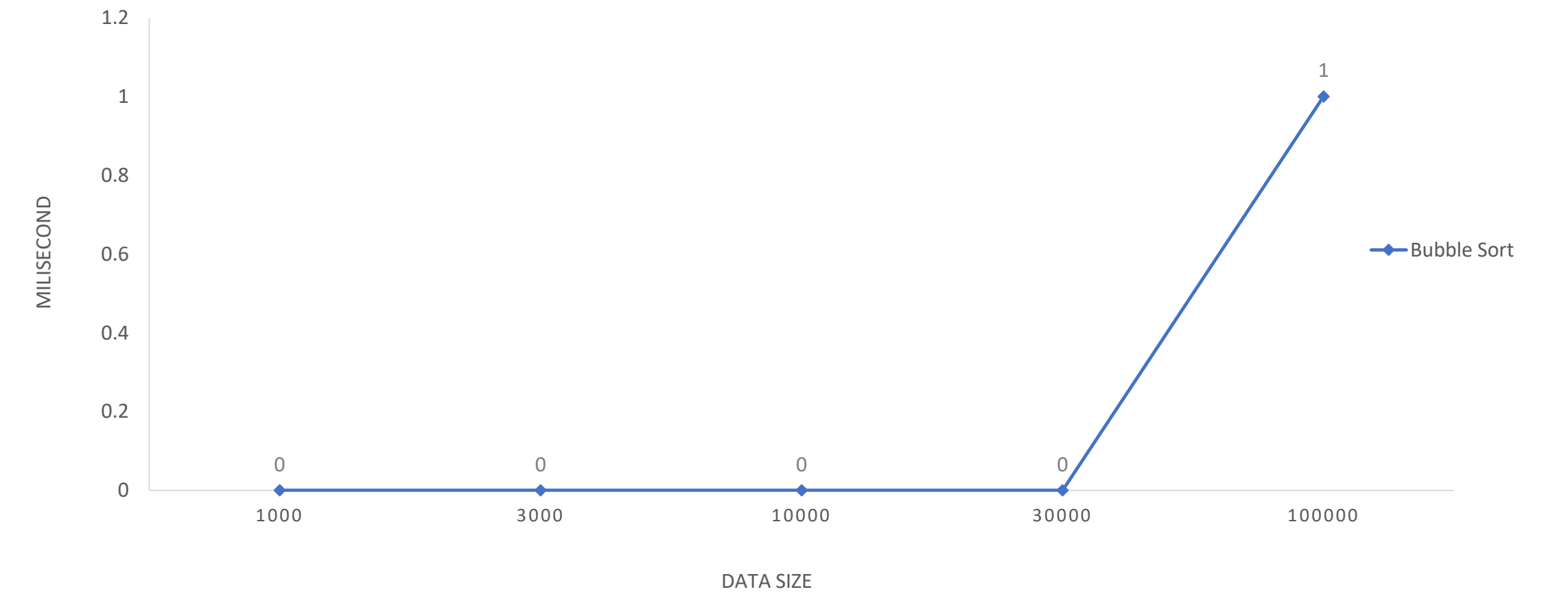
GRAPH

RANDOM DATA TYPE



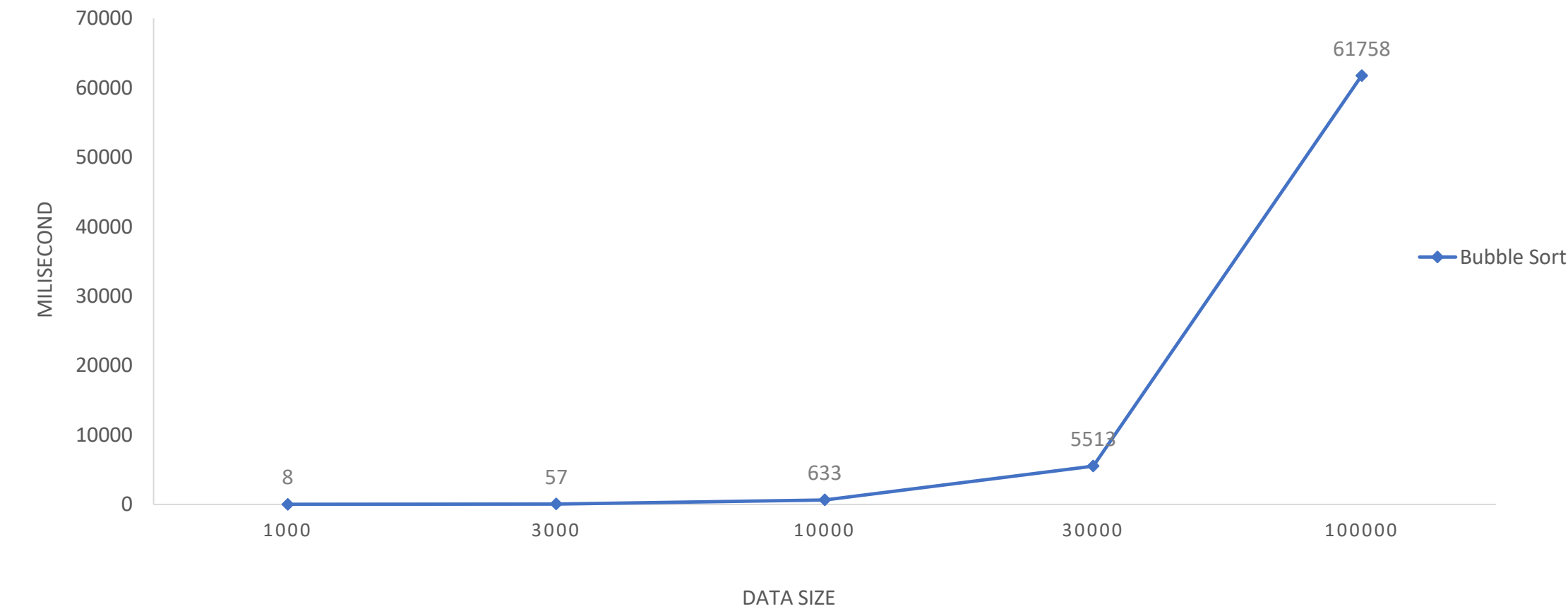
GRAPH

SORTED DATA TYPE



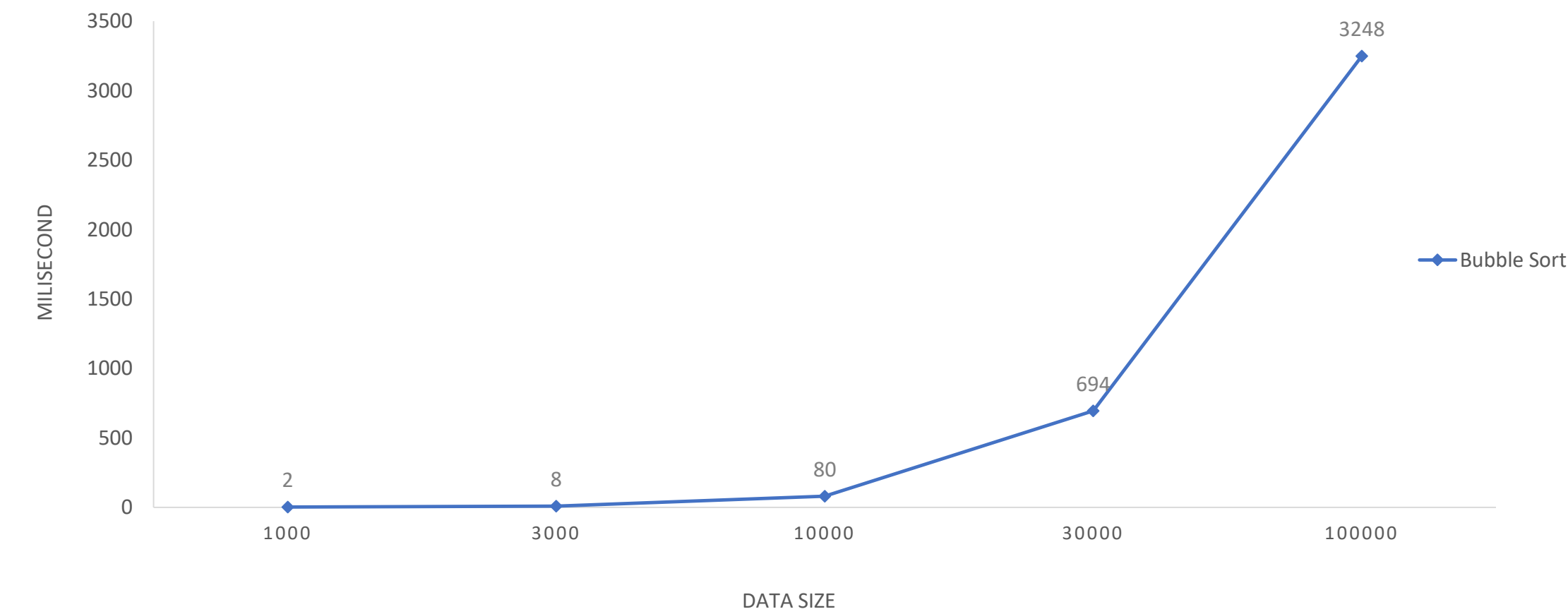
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





QUICK SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Quick sort is a *divide-and-conquer* algorithm, it will partition the array into 2 parts:

Part 1: Include elements a_1, \dots, a_i smaller than x

Part 2: Include elements a_i, \dots, a_n bigger than x

x is a pivot as we chose, after partition, the array will be separated to 3 parts:

1. $a_k < x$
2. $a_k = x$
3. $a_k > x$

Step 1: choose pivot as a random $a[k]$ in array, $\text{left} < k < \text{right}$.

$\text{pivot} = a[k]$, $l = \text{left}$, $r = \text{right}$;

Step 2: Find out and fix $a[l]$, $a[r]$ in wrong place.

a. When $a[l] < \text{pivot}$, $i++$;

b. When $a[r] > \text{pivot}$, $j--$;

c. If $l < r$ and $a[l] > a[r]$, $\text{swap}(a[l], a[r])$;

Step 3:

if $l < r$: repeat step 2;

if $l \geq r$: stop

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int j, key;

for (int i = 0; i < n; i++) {
    key = arr[i];
    j = i - 1;

    while ((j >= 0) && (arr[j] > key)) {
        arr[j + 1] = arr[j];
        j -= 1;
    }
    arr[j + 1] = key;
}
```

ALGORITHM REVIEW

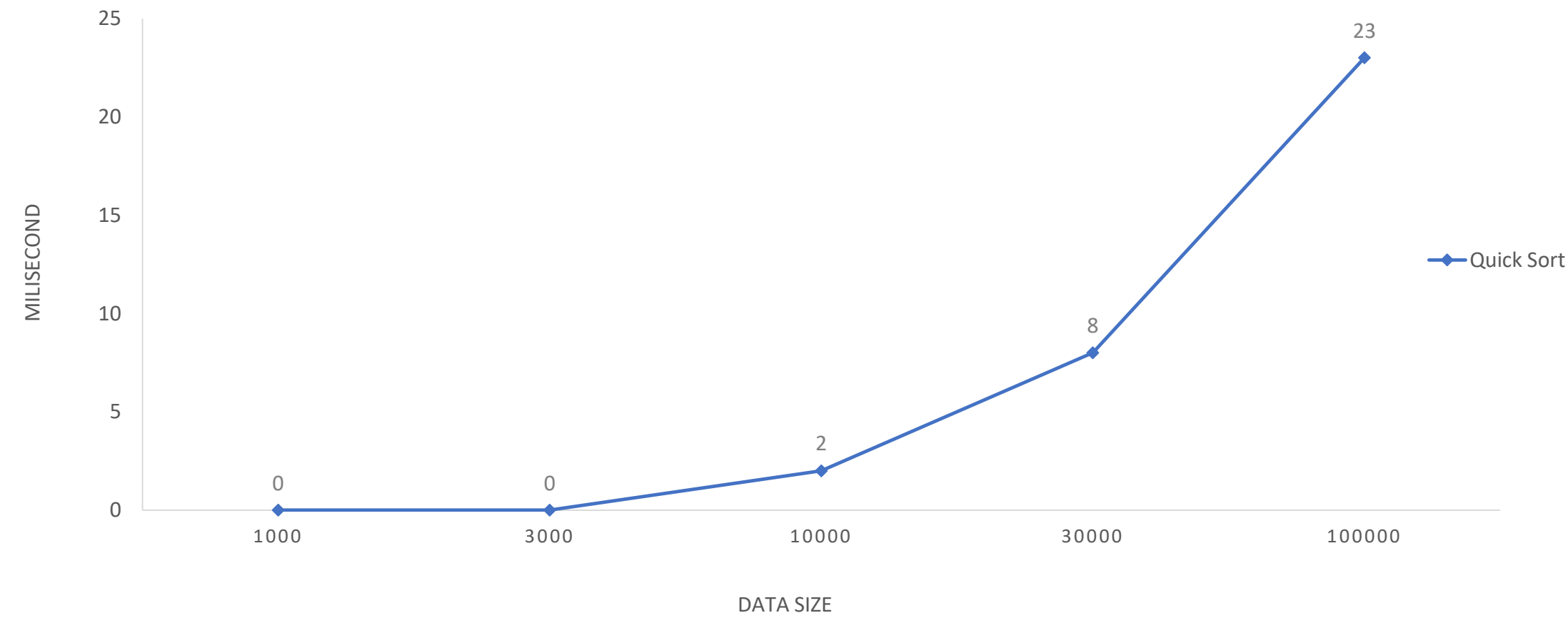
In Quick sort algorithm, we can choose any elements as a pivot. Usually we will choose a middle pivot = $(l+r)/2$.

If we choose the first or last elements in the array, the worst case would happen when the data is sorted, reverse or nearly sorted. So in these cases, we should choose the pivot as the middle element to solve this problem.

Case	Time complexity
Best	$O(n \cdot \log n)$
Worst	$O(n^2)$
Average	$O(n \cdot \log n)$

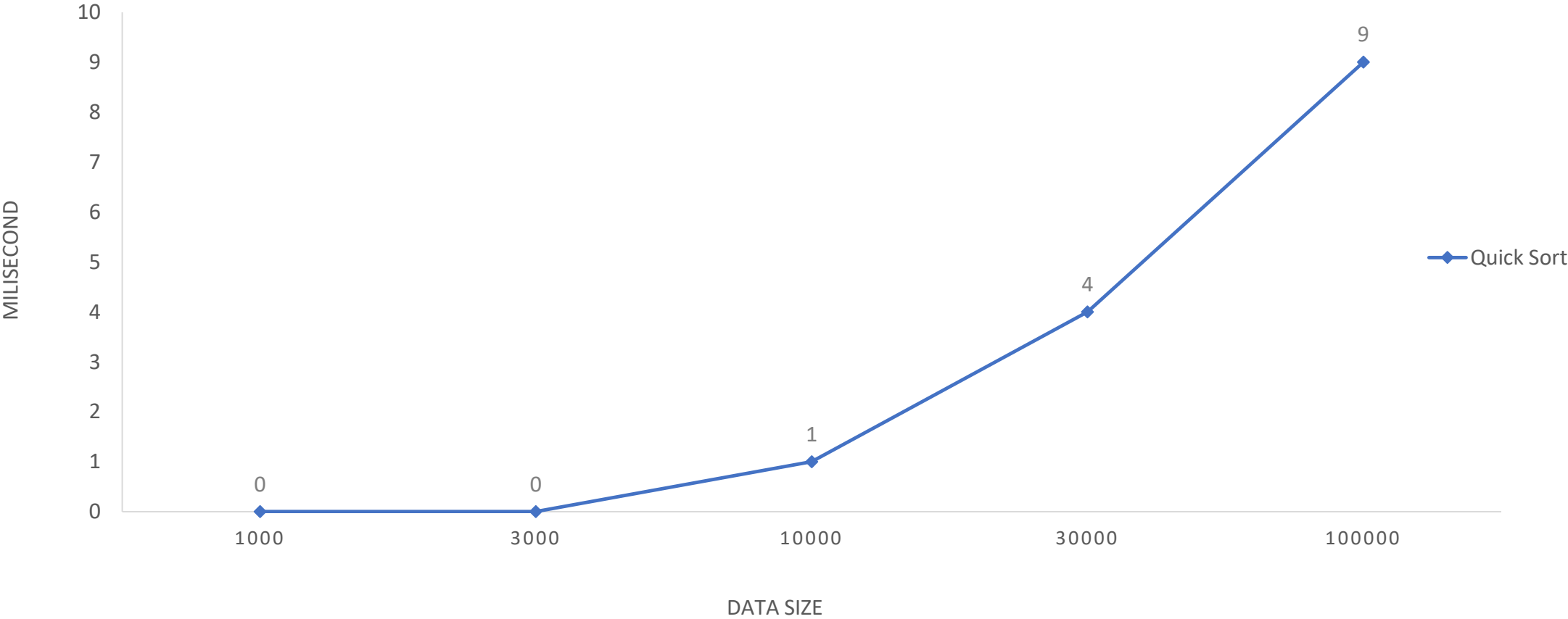
GRAPH

RANDOM DATA TYPE



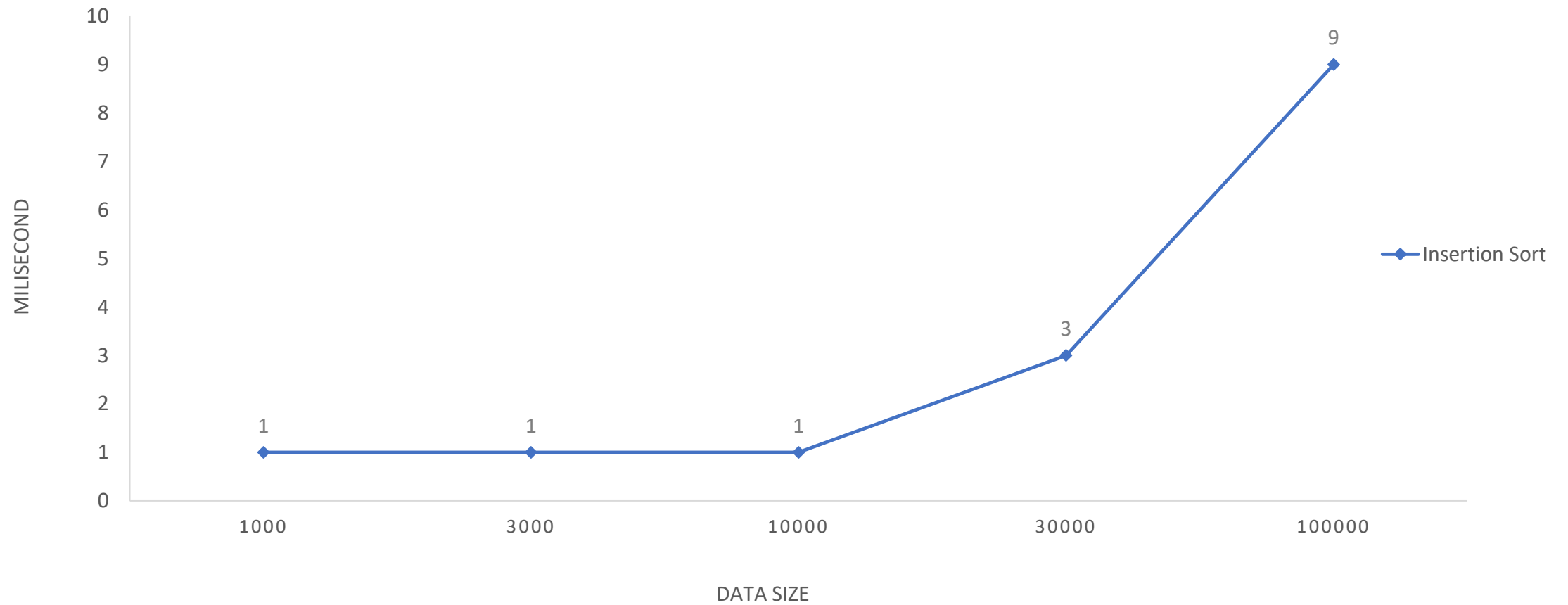
GRAPH

SORTED DATA TYPE



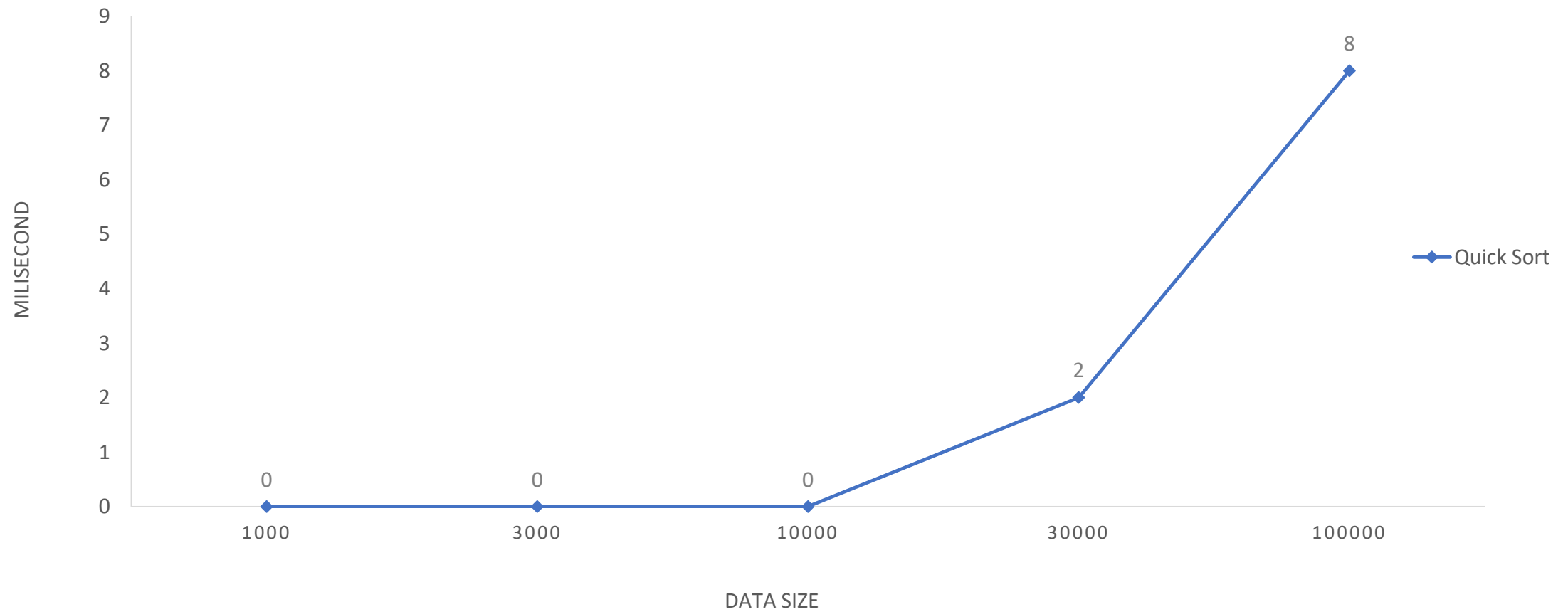
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





MERGE SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Merge sort is a divide-and-conquer algorithm based on idea of breaking down a list into several sub-lists until each sub-list contain single element.

After that, we will merge these sub lists together to create a new sorted list.

Step 1: let $m = (\text{left} + \text{right}) / 2$;

Step 2: Separate an array into 2 sub-array
array 1: from left to m
array 2: from $m+1$ to right

Step 3: if ($\text{left} < \text{right}$) do step 2;
else stop separating and move to step 4

Step 4: Merge these sub-array to create a sorted list by compare each elements in sub-arrays to place in right position

ALGORITHM

DATA STRUCTURE AND ALGORITHMS



```
void doMergeSort(int arr[], int left, int right) {  
    if (left < right) {  
        int m = left + (right - 1) / 2;  
  
        doMergeSort(arr, left, m);  
        doMergeSort(arr, m + 1, right);  
        Merge(arr, left, m, right);  
    }  
}
```

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

Merge Function

```
void Merge(int arr[], int left, int m, int right) {
    int i, j, k;
    int nr = m - left + 1;
    int nl = right - m;

    int *l, *r;
    l = new int[nr];
    r = new int[nl];

    for (i = 0; i < nr; i++) {
        l[nr] = arr[left + i];
    }

    for (j = 0; j < nl; j++) {
        r[nl] = arr[m + 1 + j];
    }
}
```

```
i = 0;
j = 0;
k = left;

while (i < nr && j < nl) {
    if (l[i] <= r[j]) {
        arr[k++] = l[i++];
    }
    else {
        arr[k++] = r[j++];
    }
}

while (i < nr) {
    arr[k++] = l[i++];
}

while (j < nl) {
    arr[k++] = r[j++];
}
}
```

ALGORITHM REVIEW

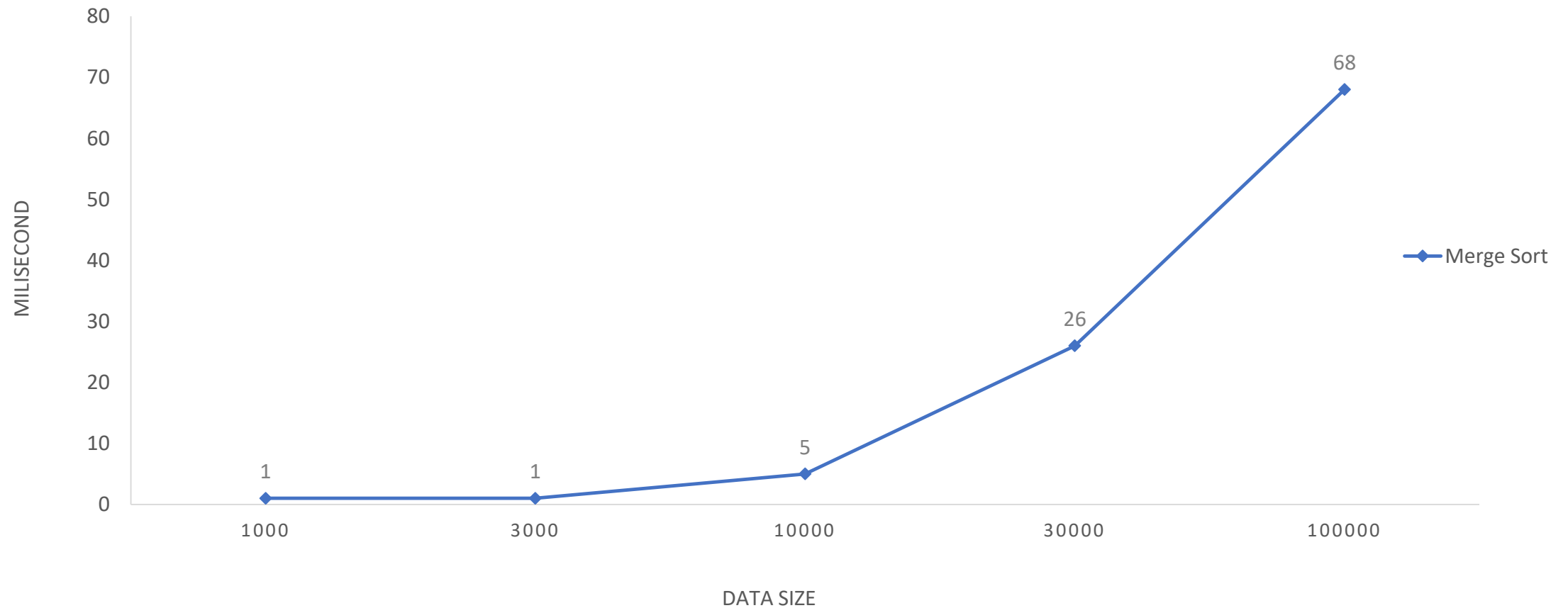
In Merge sort algorithm, the cost when we use Merge Sort always is $O(n\log_2 n)$ because it don't care about the status of this array (sorted, reverse, ...)

So this is a disadvantage of Merge sort. In real life, less people use this merge sort, people use the improved version of this merge sort called Natural Merge Sort.

Case	Time complexity
Best	$O(n\log_2 n)$
Worst	$O(n\log_2 n)$
Average	$O(n\log_2 n)$

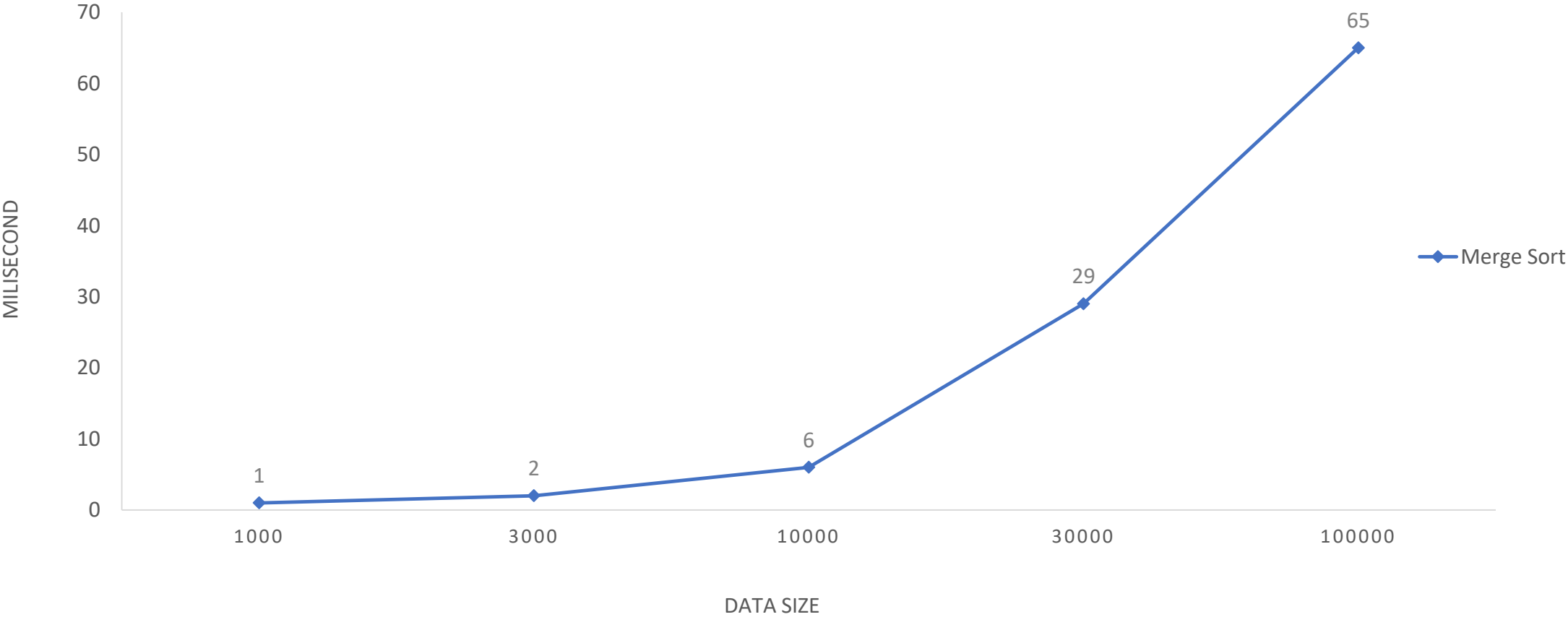
GRAPH

RANDOM DATA TYPE



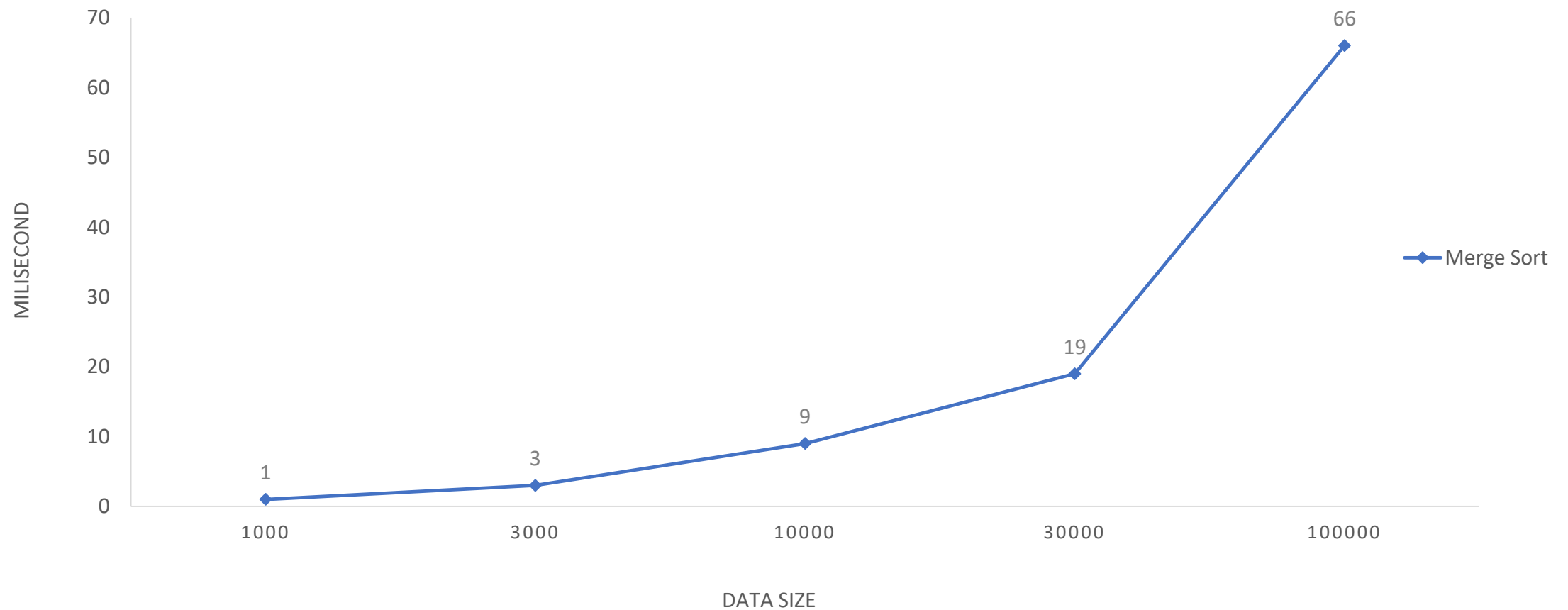
GRAPH

SORTED DATA TYPE



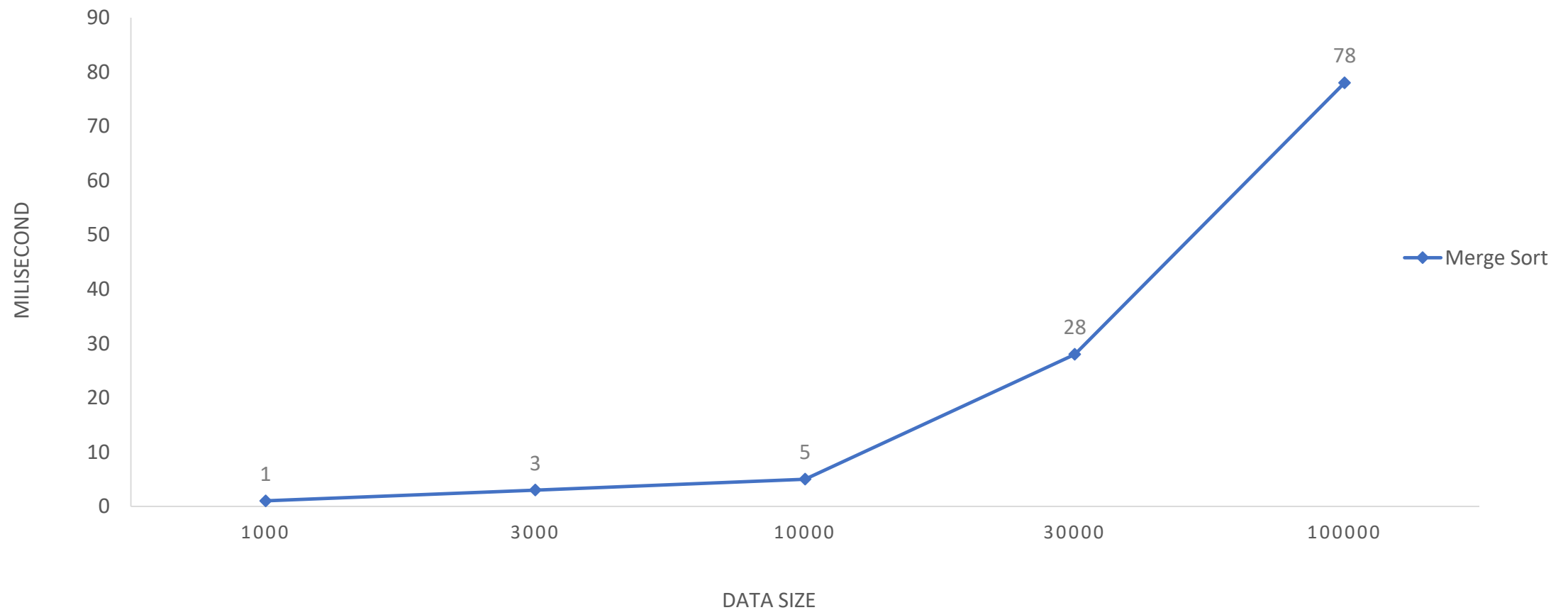
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





HEAP SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Sorting our data by using heap structure(max-heap and min-heap).

With max-heap we will have ascending data when sorting data using this structure.
And min-heap will create a descending data.

Step 1: Build a max-heap (min-heap)

Step 2: let $i = n-1$

Max-heap: swap the biggest element to the end of an array and rebuild max-heap with size $(n-1)$

Min-heap: swap the smallest element to the end of an array and rebuild min-heap with size $(n-1)$

Step 3: Continue doing step 2 until $i = 0$;
//end of array

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
for (int i = n / 2; i >= 0; i--)  
    HeapRebuild(arr, n, i);  
  
for (int i = n - 1; i >= 0; i--) {  
    swap(&arr[0], &arr[i]);  
    HeapRebuild(arr, i, 0);  
}
```

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
void HeapRebuild(int arr[], int n, int i) {  
    int largest = i;  
    int left = 2 * i + 1;  
    int right = 2 * i + 2;  
  
    if ((left < n) && (arr[left] > arr[largest]))  
        largest = left;  
    if ((right < n) && (arr[right] > arr[largest]))  
        largest = right;  
    if (largest != i)  
    {  
        swap(&arr[i], &arr[largest]);  
        HeapRebuild(arr, n, largest);  
    }  
}
```

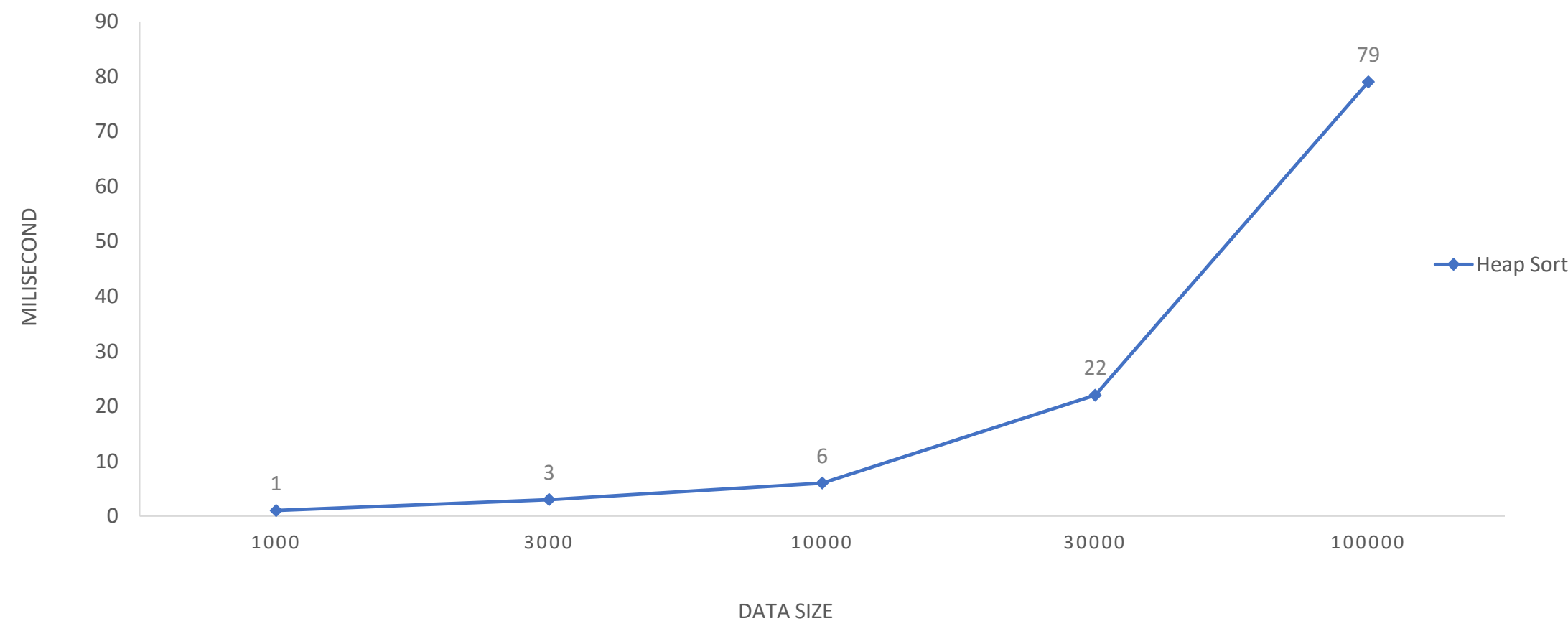

ALGORITHM REVIEW

In Heap sort algorithm, HeapRebuild has complexity $O(\log n)$, build max-heap has complexity $O(n)$ and we run HeapRebuild $(n-1)$ times in heap_sort func, therefore complexity of heap_sort function is $O(n \log n)$.

Case	Time complexity
Best	$O(n \log n)$
Worst	$O(n \log n)$
Average	$O(n \log n)$

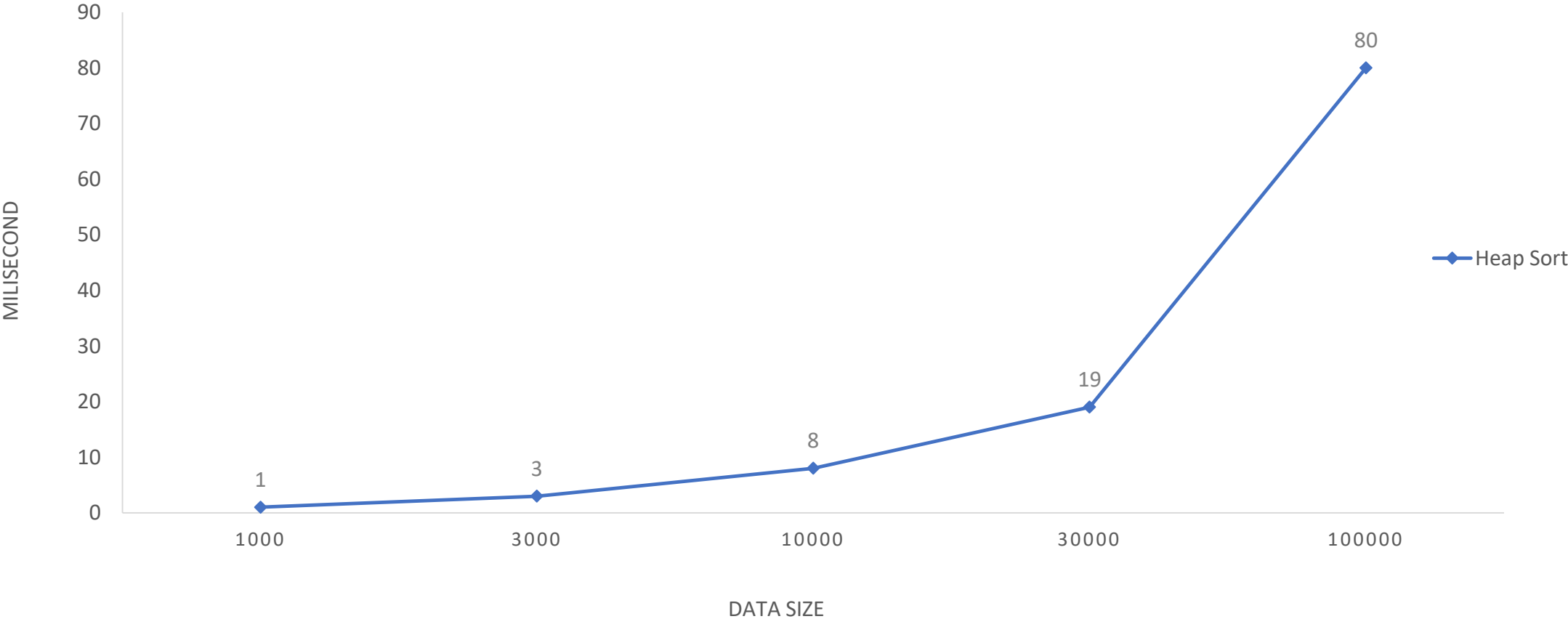
GRAPH

RANDOM DATA TYPE



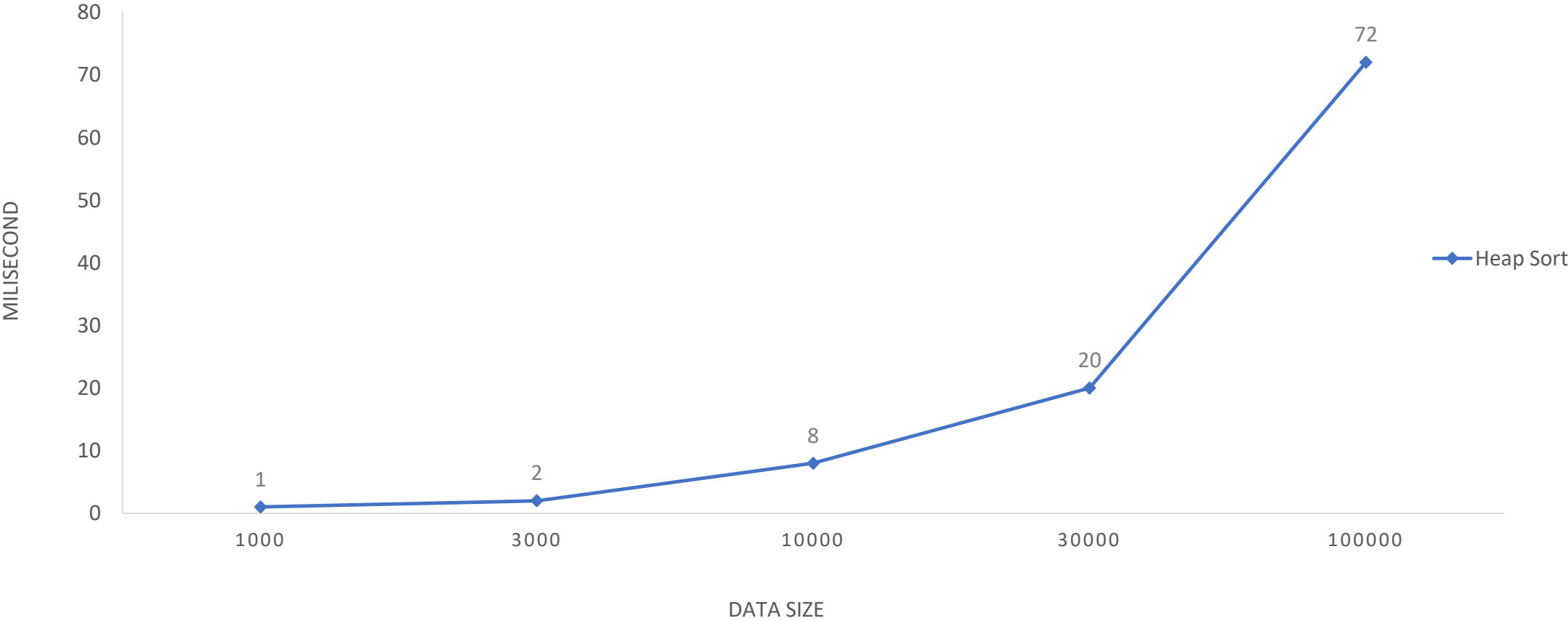
GRAPH

SORTED DATA TYPE



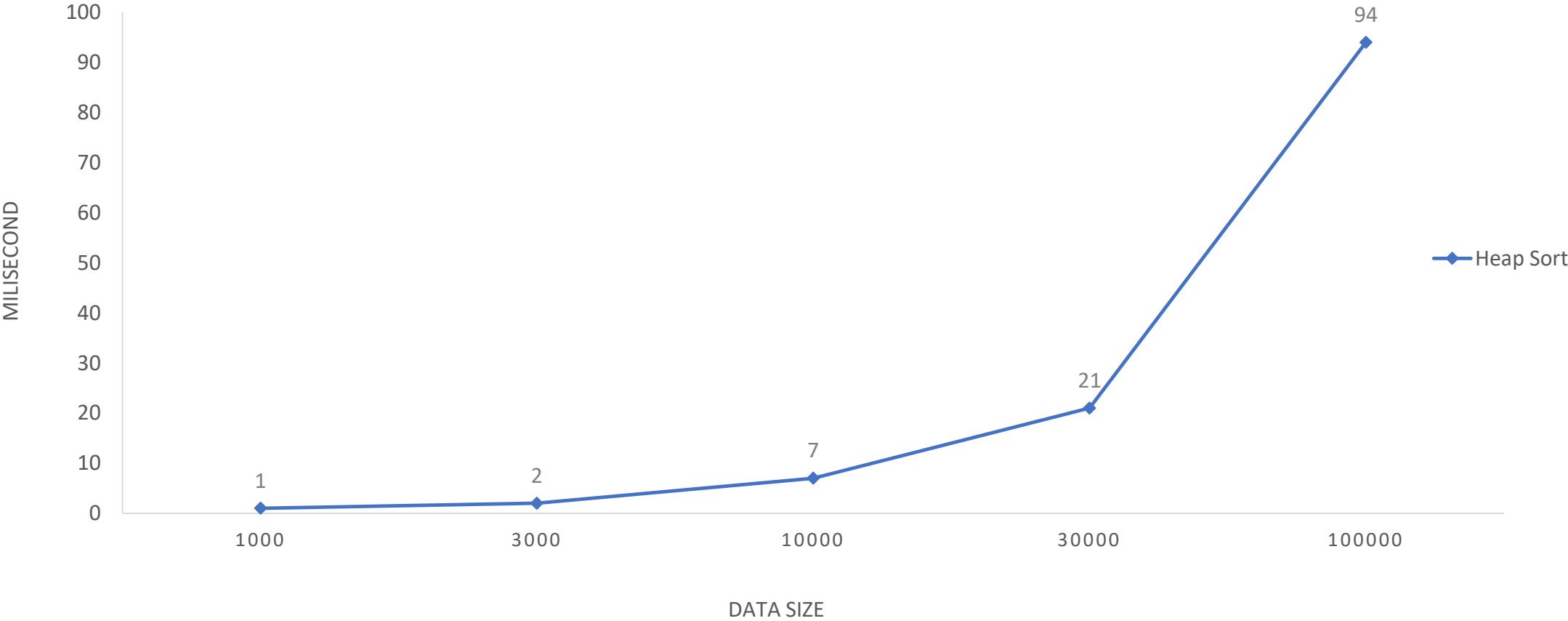
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





RADIX SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Radix sort is not a comparison based sorting algorithm, it is based on the principle of mail classification. Thus, it also has another call Postman's sort.

And the idea of Radix sort is to extend the Counting Sort algorithm to get a better time complexity.

For each digit where varies from the least significant digit to the most significant digit of a number. Sort input array using count sort algorithm according to nth digit.

Step 1: Find max in this array

Step 2: Sort the input array according to the one's digit.


Step 3: Sort according to the ten's digit.

Step 3: Continue sorting to nth digit of max value

Step 4: At the sorting nth time, we will get the sorted array.

ALGORITHM

DATA STRUCTURE AND ALGORITHMS



```
int max = findMax(arr, n);

for (int div = 1; (max / div) > 0; div *= 10) {
    subCountSort(arr, n, div);
}
```


ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
void subCountSort(int arr[], int n, int div) {
    int i;
    int count[10] = { 0 };
    int *out;
    out = new int[n];

    for (i = 0; i < n; i++) {
        count[(arr[i] / div) % 10]++;
    }

    for (i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (i = n - 1; i >= 0; i--) {
        out[count[(arr[i] / div) % 10] - 1] = arr[i];
        count[(arr[i] / div) % 10]--;
    }

    for (i = 0; i < n; i++) {
        arr[i] = out[i];
    }
}
```

ALGORITHM REVIEW

With an array have n elements and each elements have k digits. The radix sort will do k times to place the number into the sub array following it digits. So the cost for this is $O(2nk) = O(n)$.

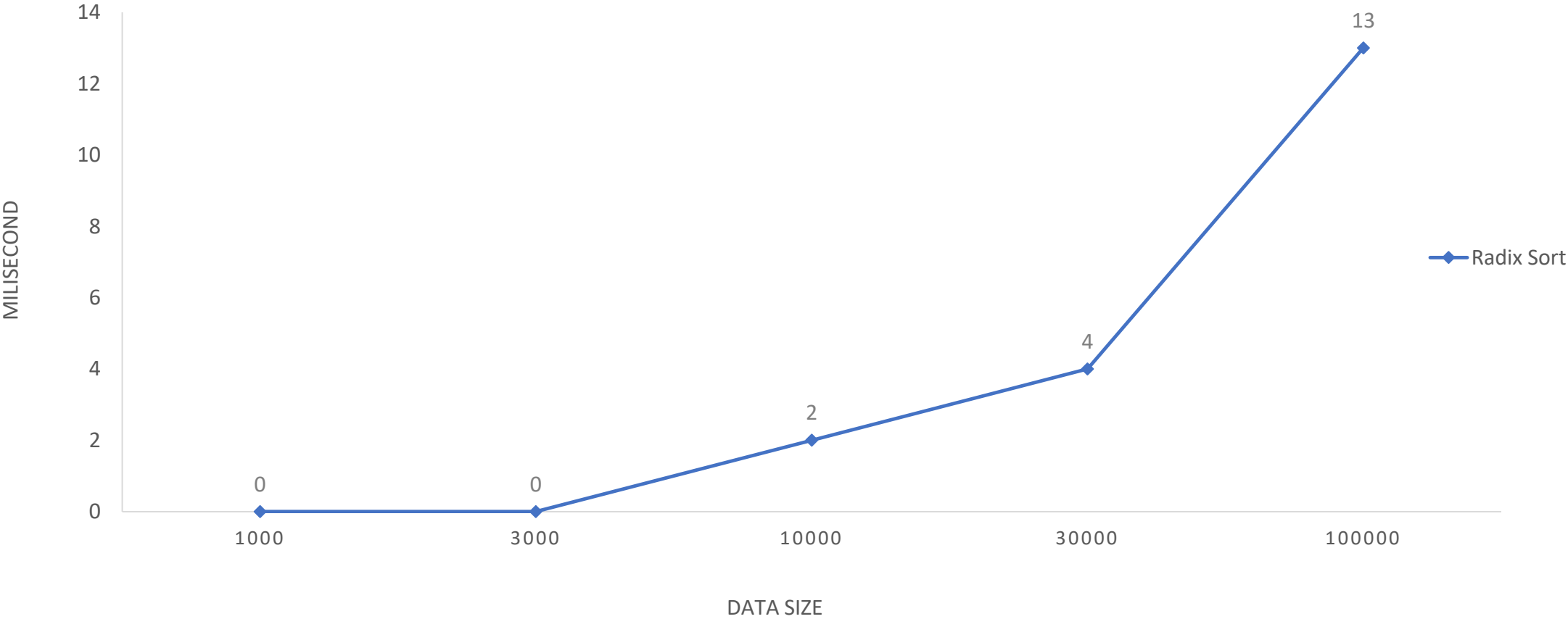
The complexity of this algorithm is linear, so it is efficiency when we sort the huge array.

This algorithm do not have worst case and best case. Every element will be placed into the array with the same cost if they have same digits.

Case	Time complexity
Best	$O(d(n+k))$
Worst	$O(d(n+k))$

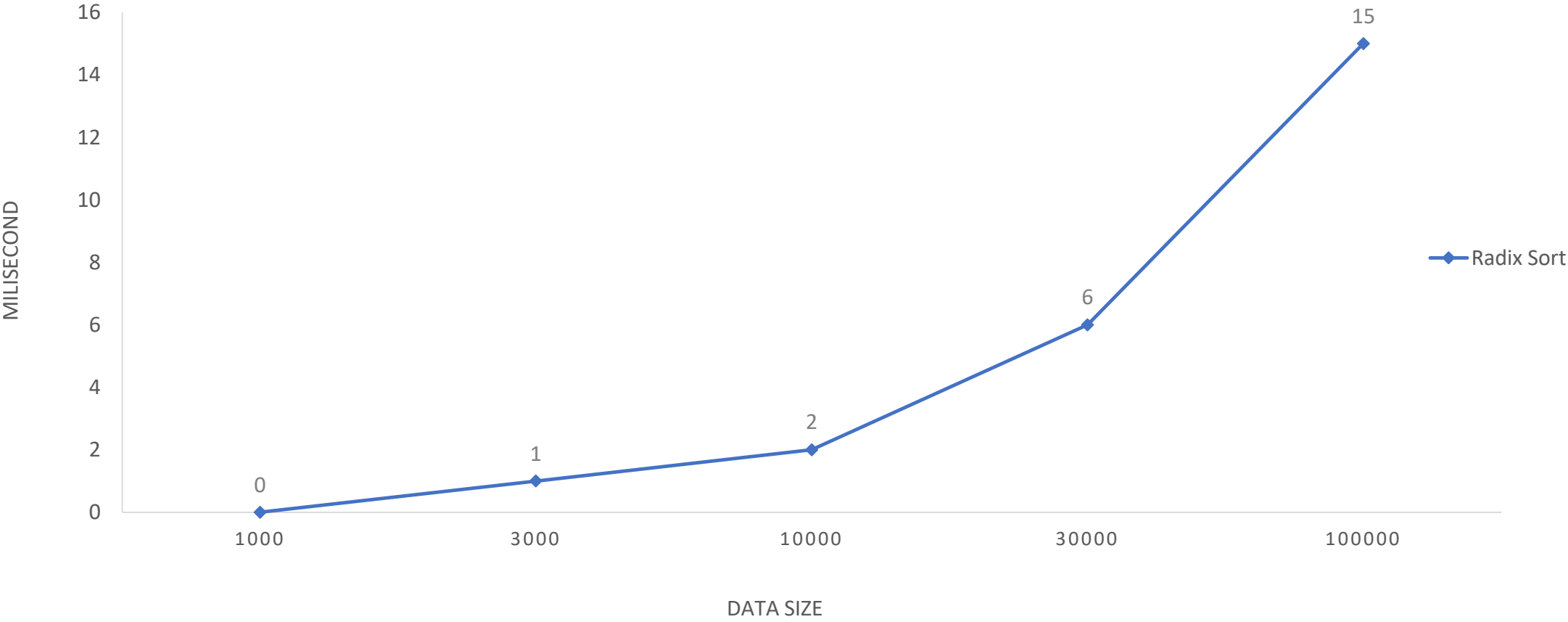
GRAPH

RANDOM DATA TYPE



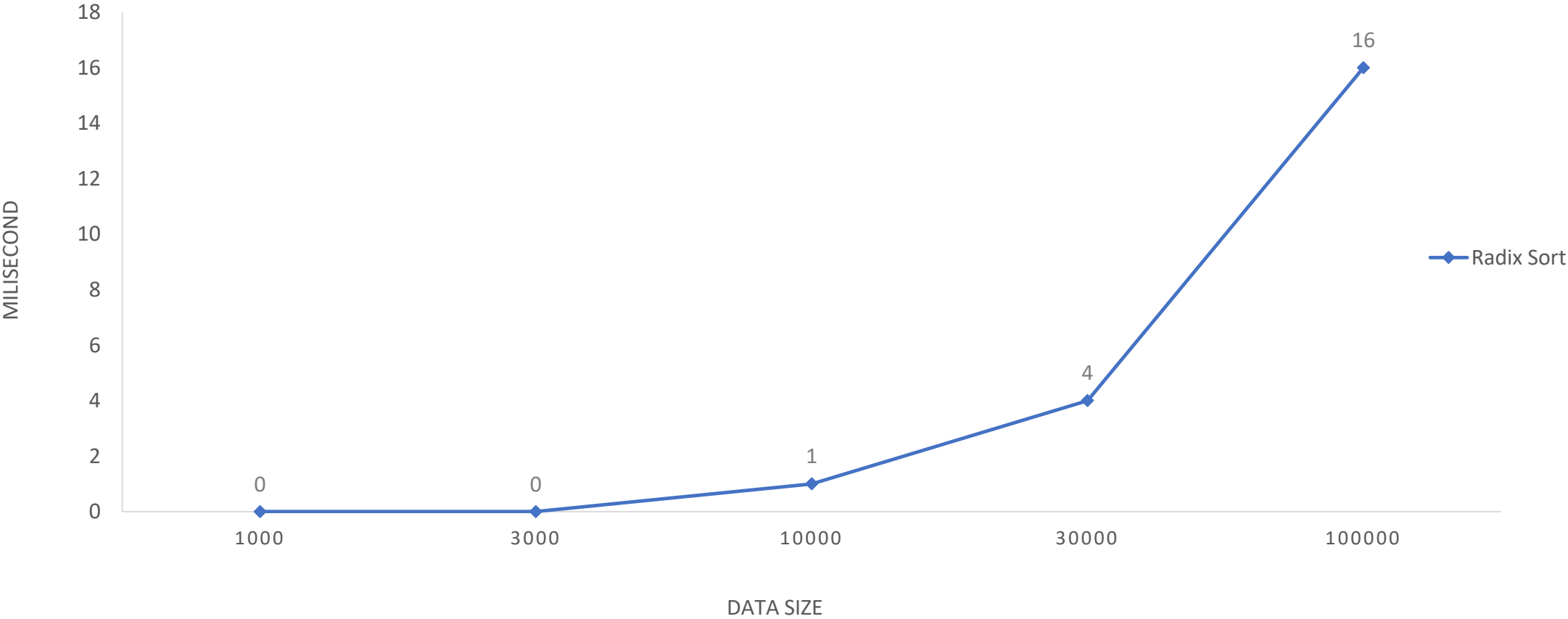
GRAPH

SORTED DATA TYPE



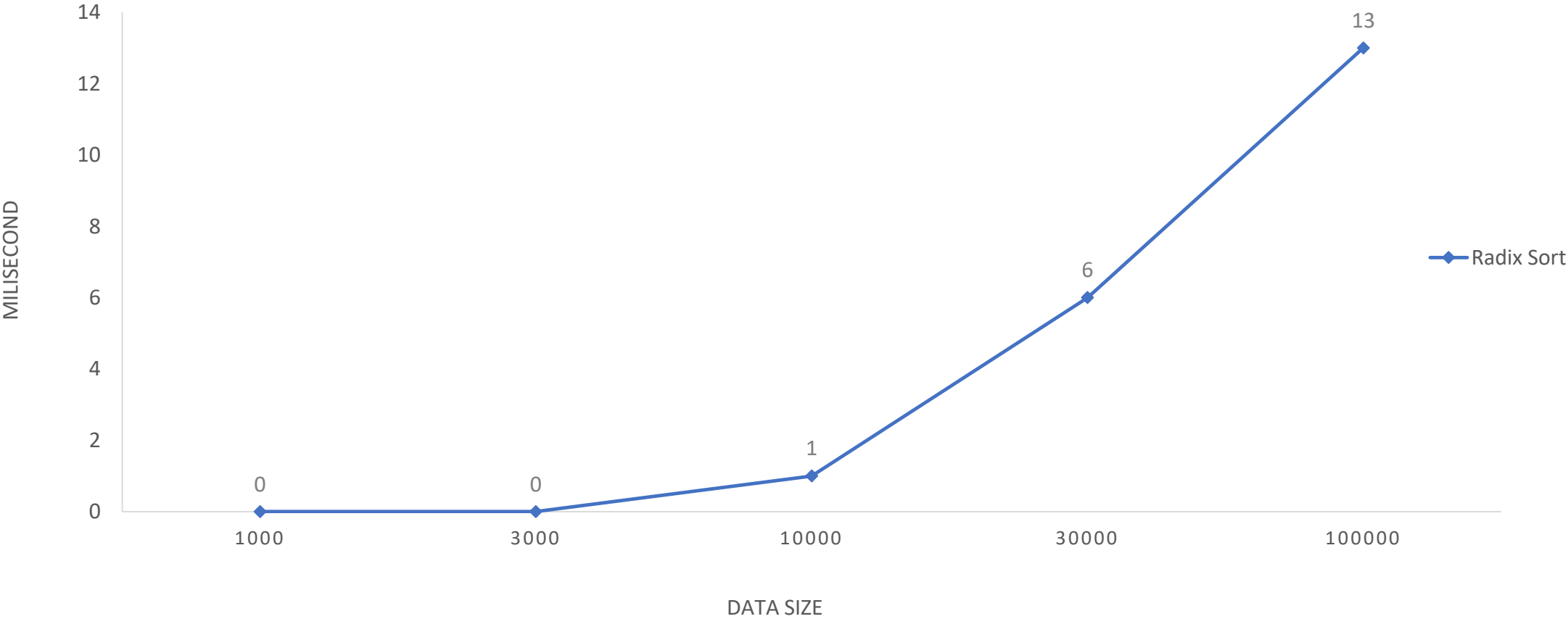
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





COUNTING SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

The basic idea of Counting sort is finding the max value in the array and create a new array with max elements.

Purpose to count the appearance times of $a[i]$ value in the array.

Following the new created array, we will have a sorted array.

Step 1: Find max and create new array "count array" with max elements

Step 2: Counting the appearance of each value in unsorted array in "count array"

Step 3:

Add up the values in the populated "count array"
Create a new array to store our sorted array

Step 4: Copy to our old array with sorted data from new created array.

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int *tmp = new int[n];

int max(arr[0]), min(arr[0]);

for (int i = 1; i < n; i++) {
    if (arr[i] > max)
        max = arr[i];
    else if (arr[i] < min)
        min = arr[i];
}
```

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int size = max - min + 1;

int *count = new int[size]();

for (int i = 0; i < size; i++)
    count[arr[i] - min]++;

for (int i = 1; i < size; i++) {
    count[i] += count[i - 1];
}

for (int i = 0; i < n; i++) {
    tmp[count[arr[i] - min] - 1] = arr[i];
    count[arr[i] - min]--;
}

for (int i = 0; i < n; i++) {
    arr[i] = tmp[i];
}
```

ALGORITHM REVIEW

Counting sort operates only on integers.

Counting sort assumes that you know the range of your input (the integers you are trying to sort).

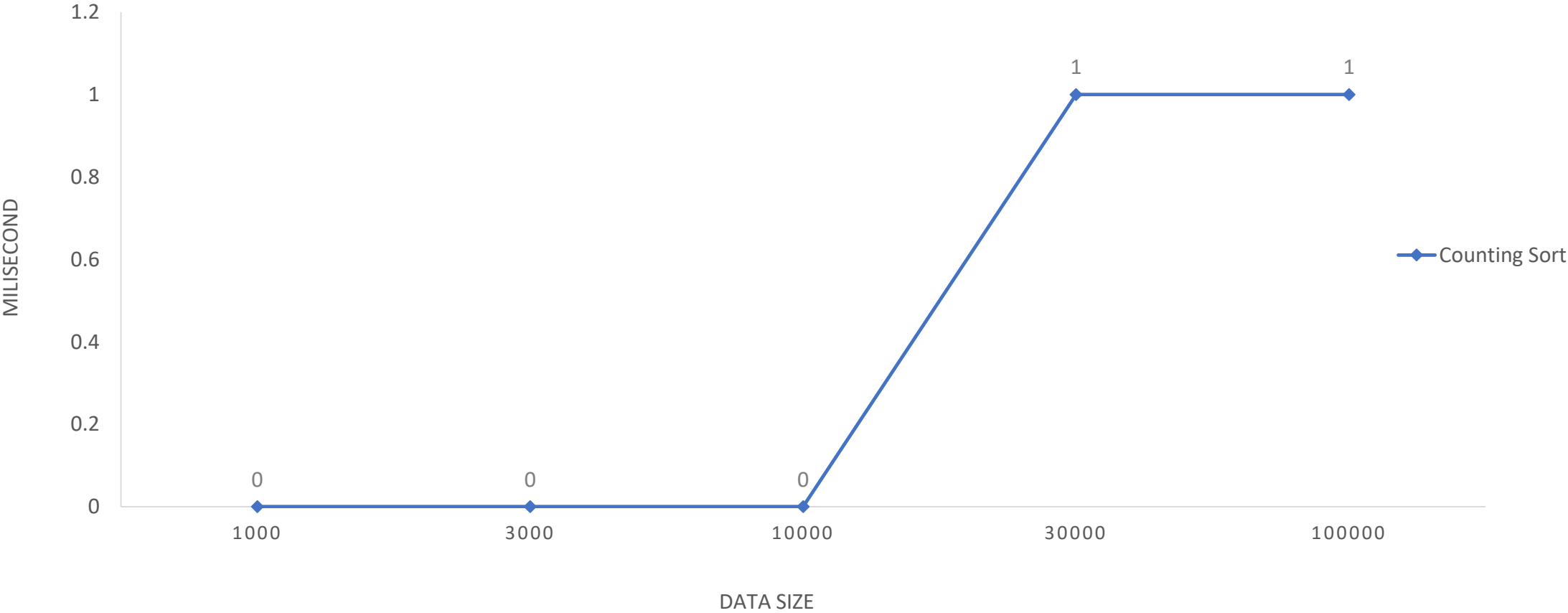
Counting sort works best if the range of the integers to be sorted isn't too wide, that is to say, not greater than the number of items to be sorted.
Generally, it works at its best on smaller integers.

******If we only have 5 elements to sort but the range is too wide (between 0 and 10000), counting sort wouldn't work well, since it has to create "count" array.

Case	Time complexity
Best	$O(n+k)$
Worst	$O(n+k)$

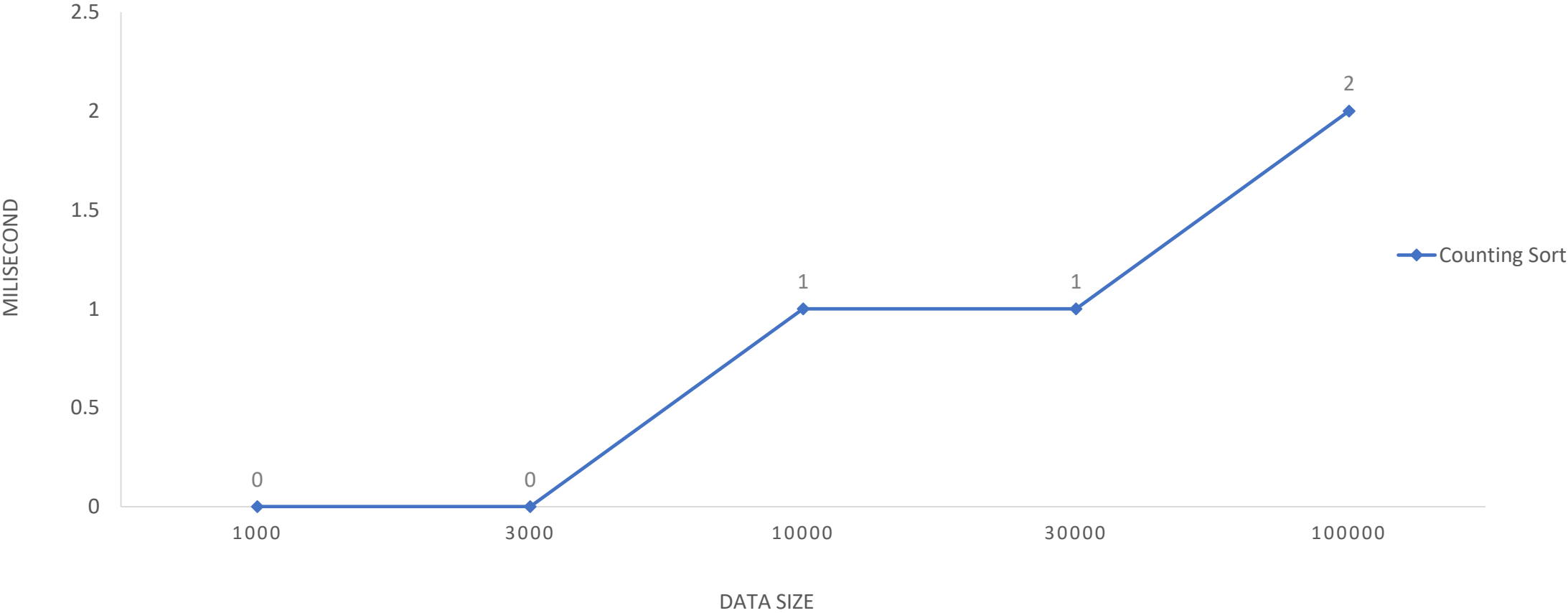
GRAPH

RANDOM DATA TYPE



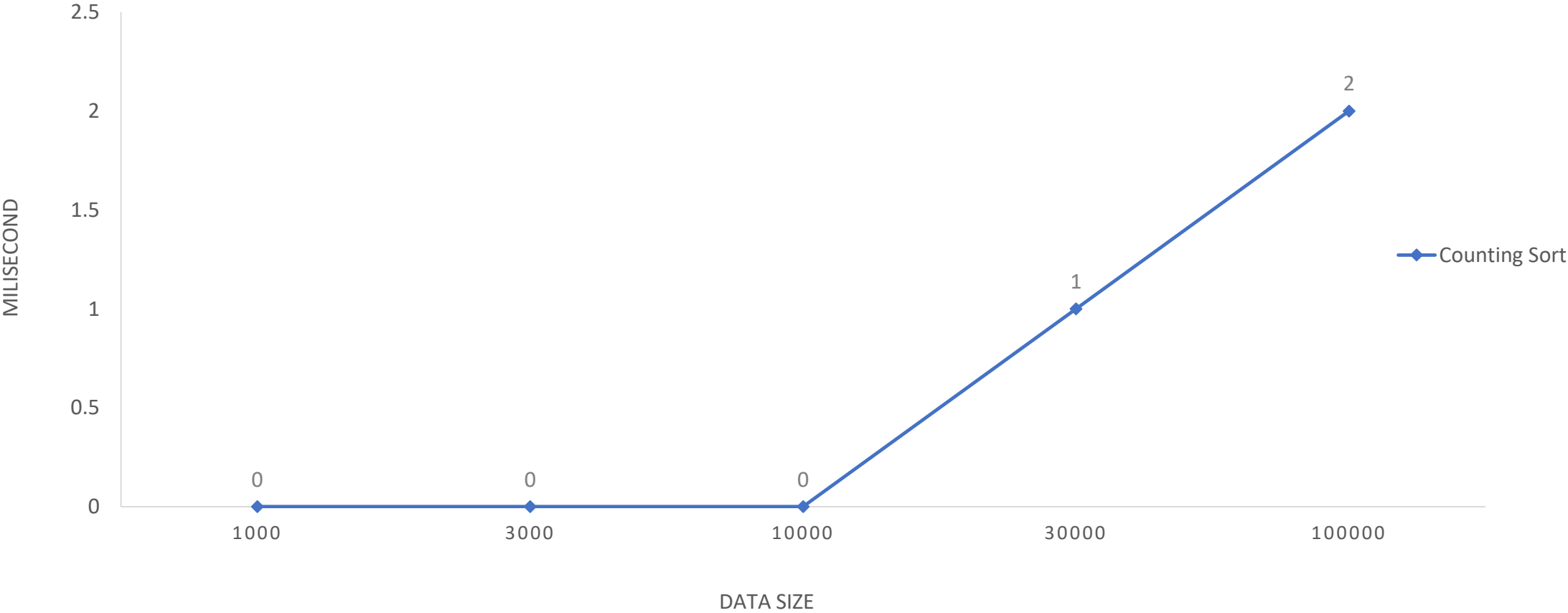
GRAPH

SORTED DATA TYPE



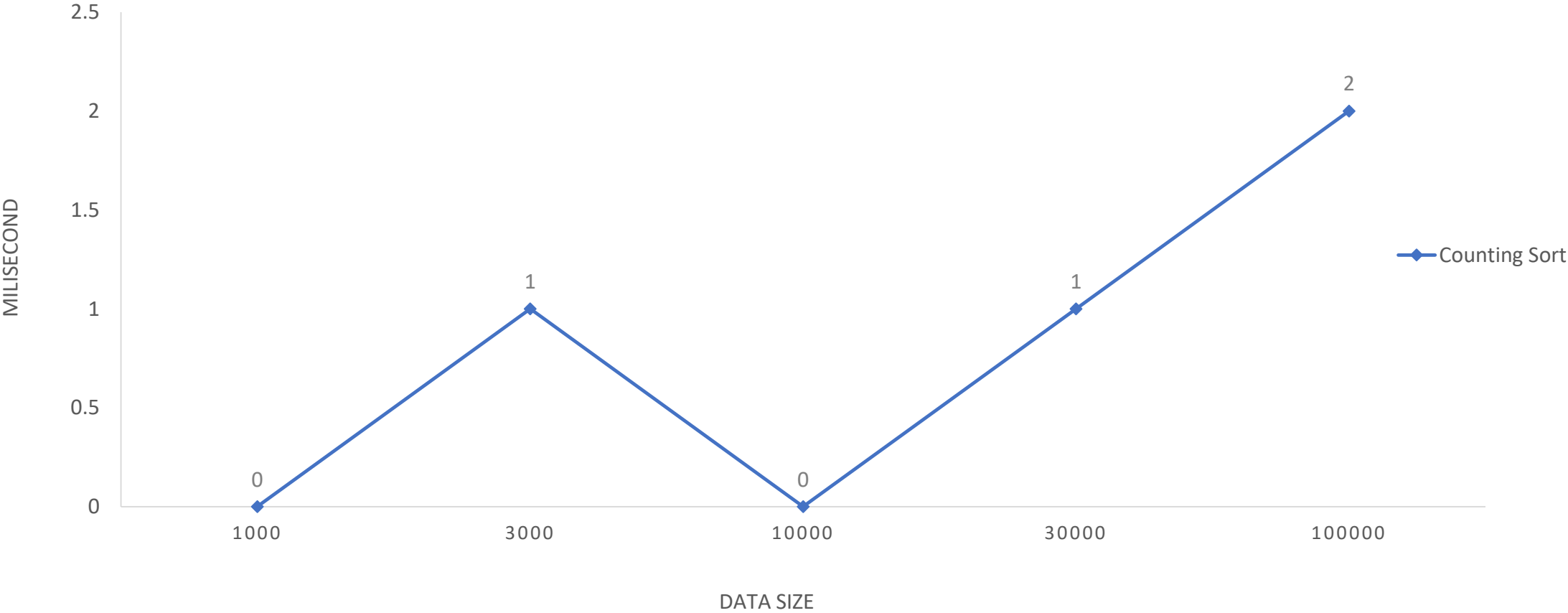
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





SHELL SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Shell sort is an improved version of Insertion sort. The main idea of this algorithm is dividing the array into sub arrays with k apart.

The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements, it can move some out-of-place elements into position faster than a simple nearest neighbor exchange.

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
for (int mid = n / 2; mid > 0; mid /= 2) {  
    for (int i = mid; i < n; i++) {  
        int tmp = arr[i];  
  
        int j = i;  
        while (j >= mid && arr[j - mid] > arr[j]) {  
            arr[j] = arr[j - mid];  
            j -= mid;  
        }  
        arr[j] = tmp;  
    }  
}
```

ALGORITHM REVIEW

Analyzing Shell sort is very complicated, and efficiency of Shell sort depend on the length is chosen.

$O(n^2)$ (worst known worst case gap sequence)

$O(n \log_2 n)$ (best known worst case gap sequence)

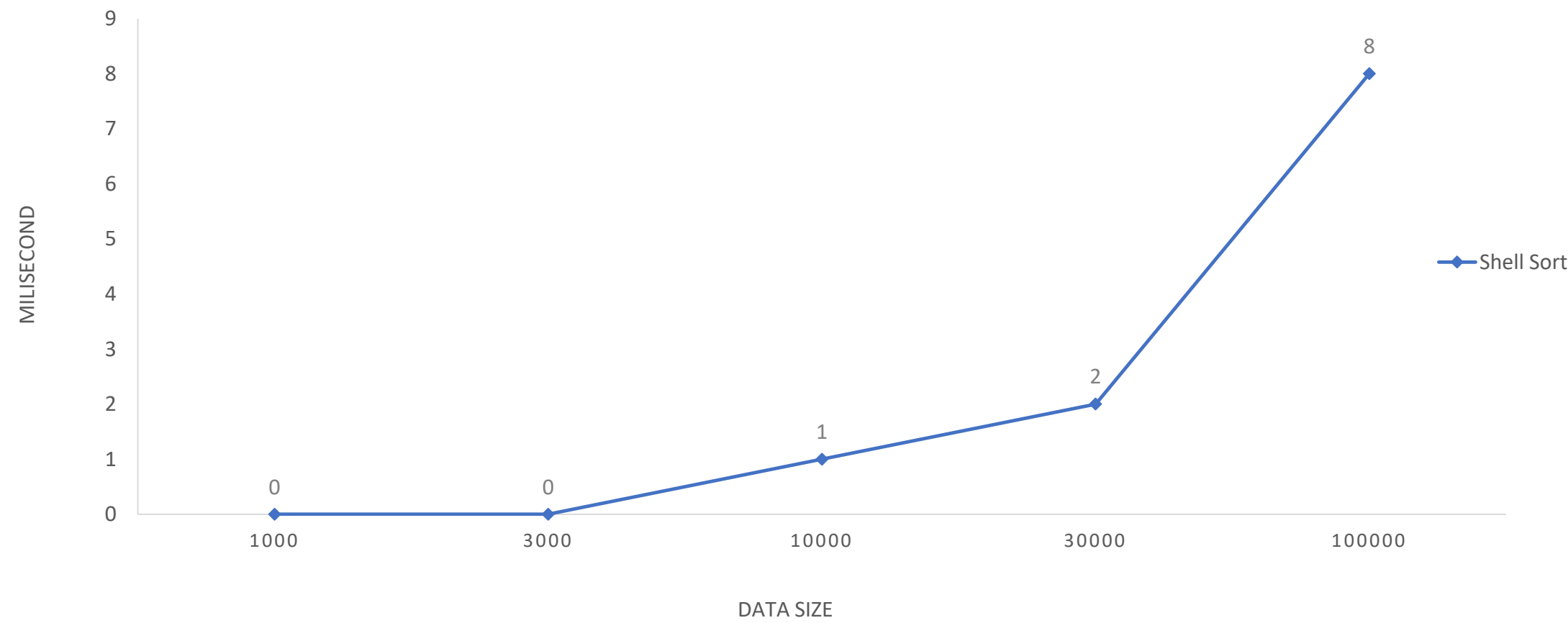
$O(n \log n)$ (best-case performance - most gap sequences)

$O(n \log_2 n)$ (best known worst-case gap sequence)

Case	Time complexity
Best	$O(n^2)$
Worst	$O(n \log n)$

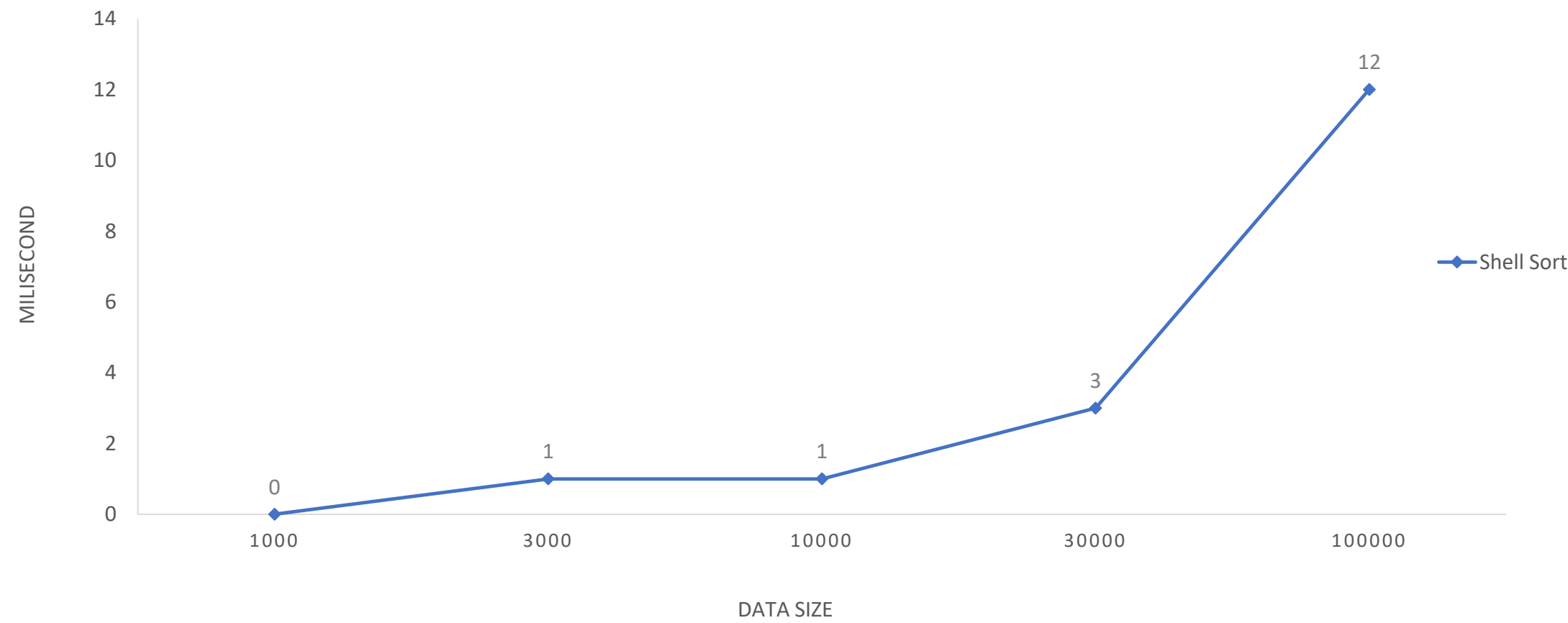
GRAPH

RANDOM DATA TYPE



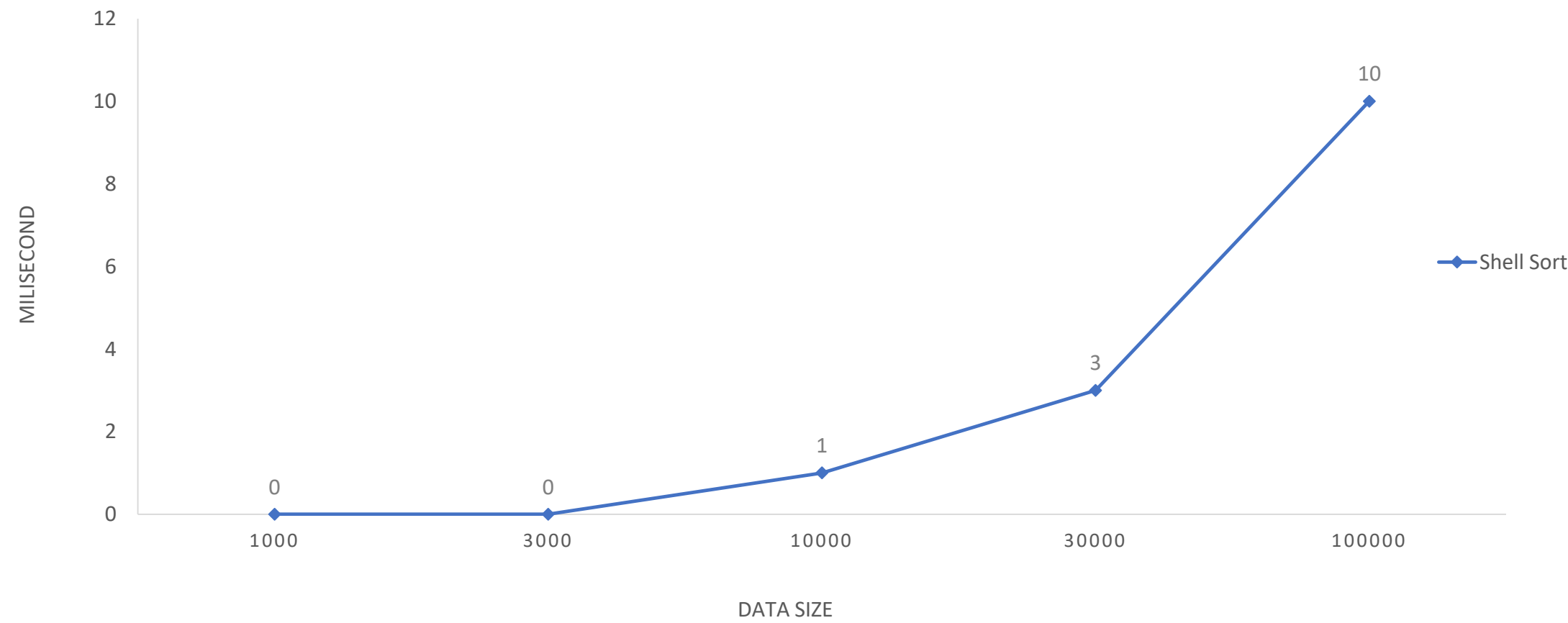
GRAPH

SORTED DATA TYPE



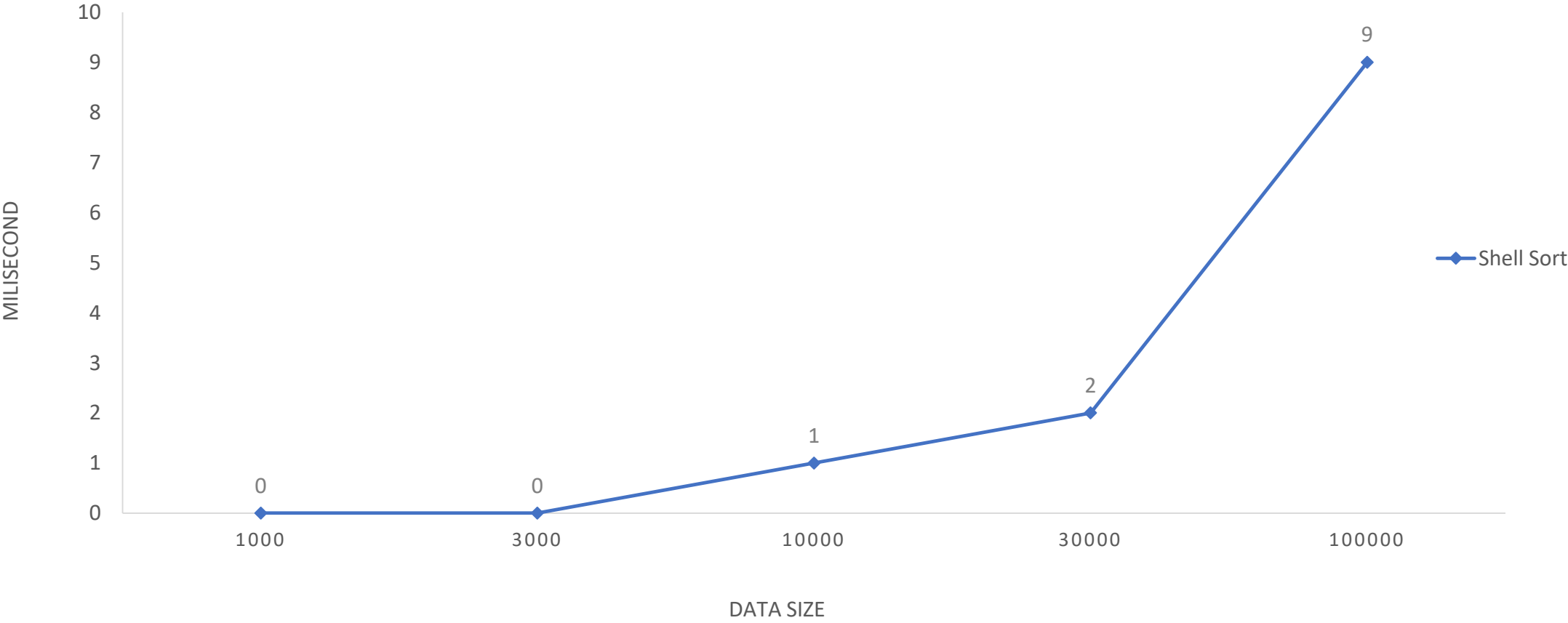
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





SHAKER SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Shaker sort is an improved version of Bubble sort.
After bringing the smallest to the top of the array,
then we will bring the biggest to the end of the
array.

It try to fix the disadvantage of Bubble sort.

Step 1: let left = 0, right = n- 1

Step 2: run i from left to right
if ($a[i] > a[i+1]$) swap
update right

Step 3: run i from right to left
if($a[i] < a[i+1]$) swap
update left

Step 4: if (left < right) continue step 2;
else stop

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int left(0), right(n - 1);
int k = 0;

while (left < right) {
    for (int i = left; i < right; i++) {
        if (arr[i] > arr[i + 1]) {
            swap(&arr[i], &arr[i + 1]);
            k = i;
        }
    }
    right = k;

    for (int i = right; i > left; i--) {
        if (arr[i] < arr[i + 1]) {
            swap(&arr[i], &arr[i + 1]);
            k = i;
        }
    }
    left = k;
}
```

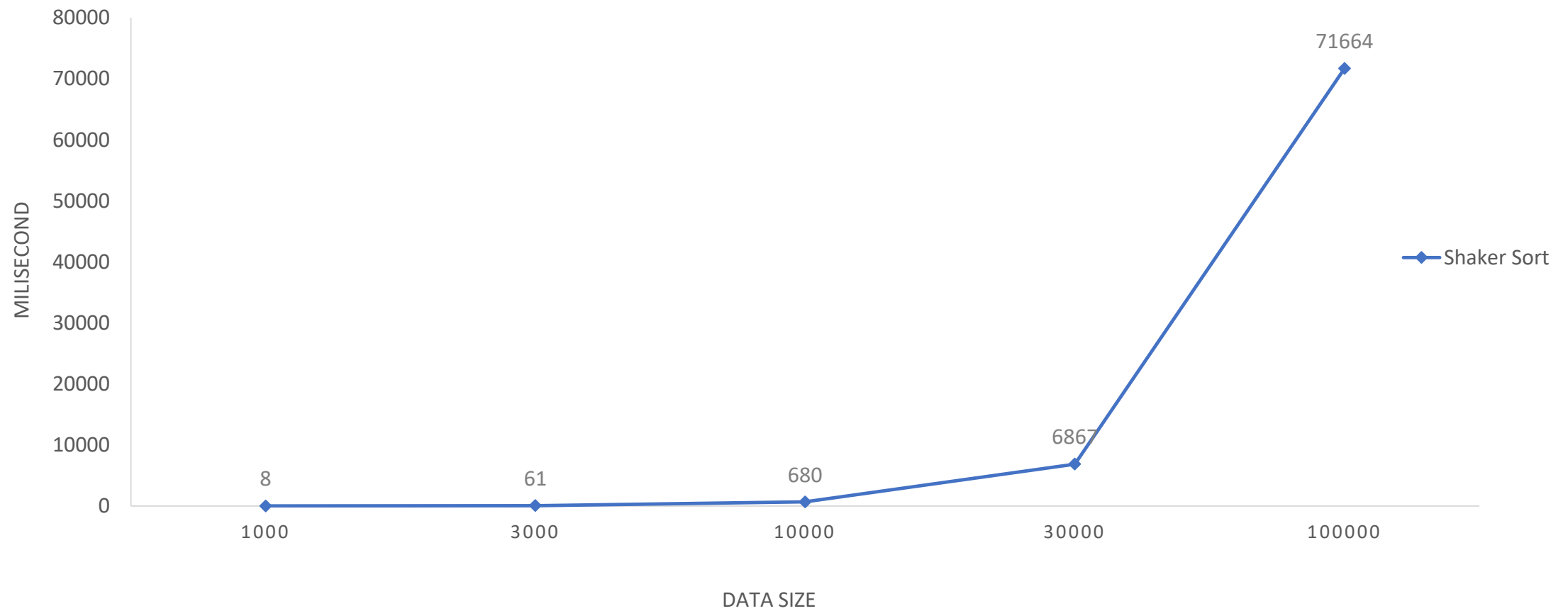
ALGORITHM REVIEW

The complexity of the shaker sort in big O notation is for both $O(n^2)$ in the worst case and the average case, but it becomes closer to $O(n)$ if the list is mostly ordered before applying the sorting algorithm.

Case	Time complexity
Best	$O(n)$
Average	$O(n^2)$
Worst	$O(n^2)$

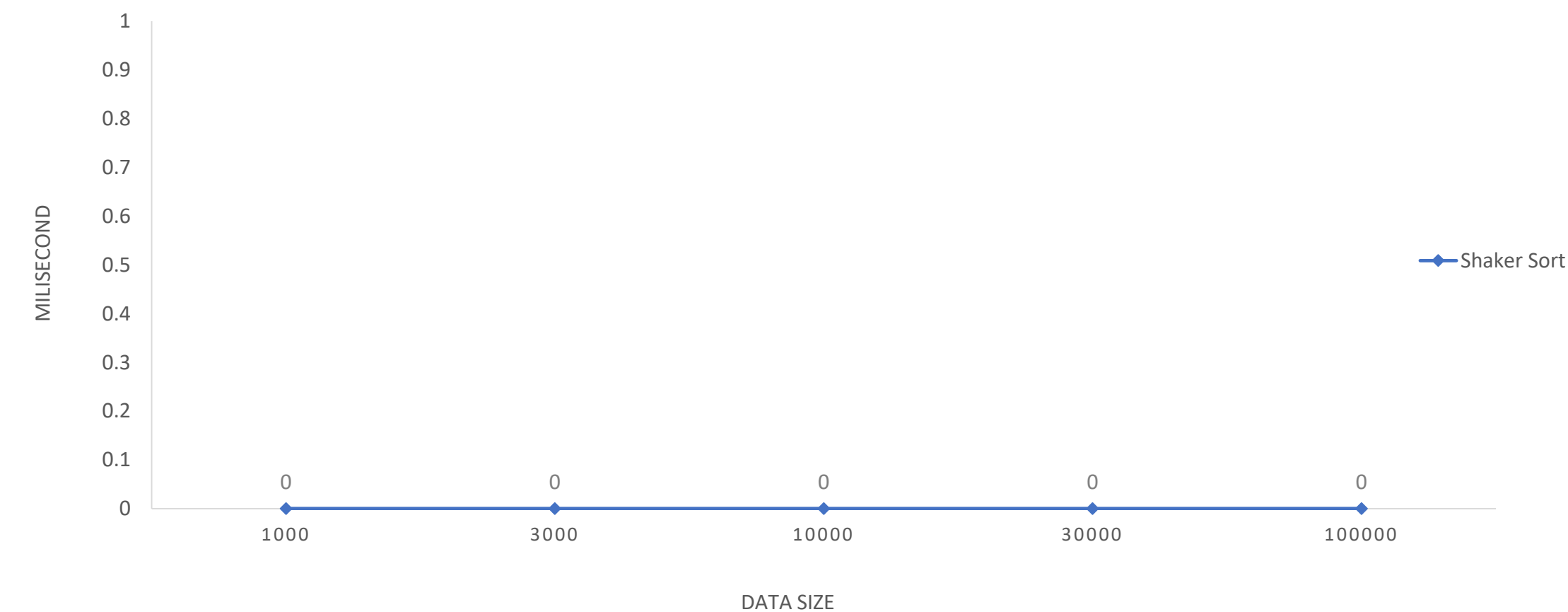
GRAPH

RANDOM DATA TYPE



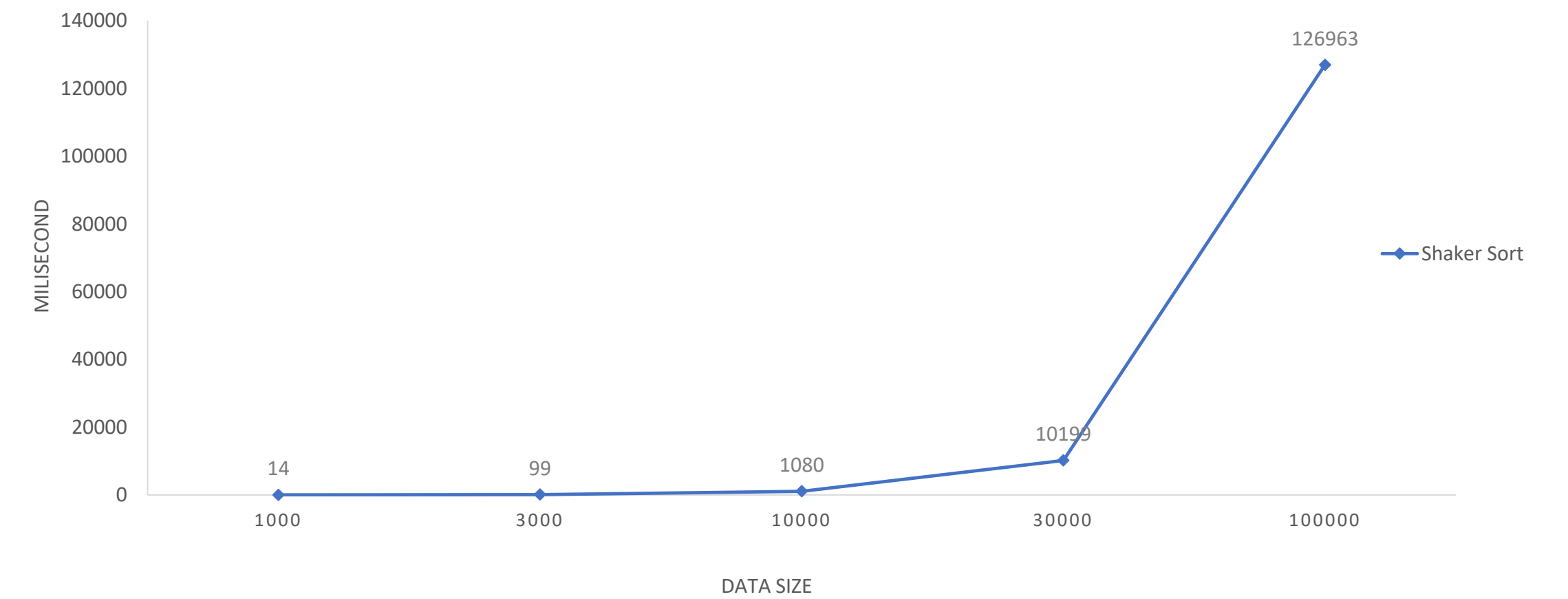
GRAPH

SORTED DATA TYPE



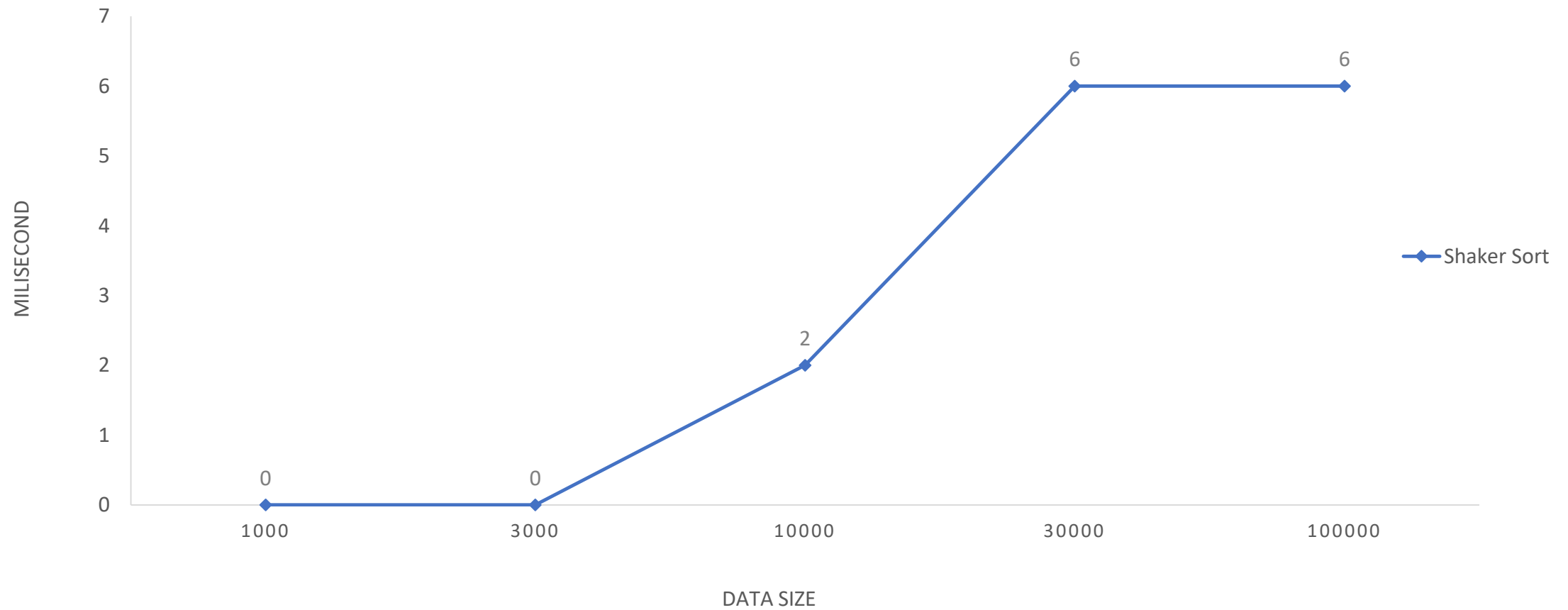
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





BINARY-INSERTION SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

I D E A

Binary-insertion sort is an improved version of Insertion sort.

Binary insertion sort employs a binary search to determine the correct location to insert new elements.

Step 1: let $i = 2$; //suppose that we already had part $a[1]$ is sorted.

Step 2: $x = a[i]$, Find the position in sorted part $a[1]$ to $a[i-1]$ to insert $a[i]$ into it.

Step 3:

Using Binary search to find location of x in sorted part array.

Displace all elements from $a[\text{location}]$ to $a[i-1]$ to right 1 unit to take place for $a[i]$ get in.

Step 4: $a[\text{location}] = x$

Step 5: $i = i + 1$;

If $i \leq n$: Repeat step 2

Else: Stop

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int i, location, j, k, selected;
for (i = 1; i < n; ++i)
{
    j = i - 1;
    selected = arr[i];

    location = binarySearch(arr, selected, 0, j);

    while (j >= location)
    {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = selected;
}
```

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

```
int binarySearch(int arr[], int select, int left, int right) {  
    while (left <= right) {  
        int m = left + (right - left) / 2;  
        if (select == arr[m])  
            return m + 1;  
        if (select < arr[m])  
            right = m - 1;  
        else  
            left = m + 1;  
    }  
}
```

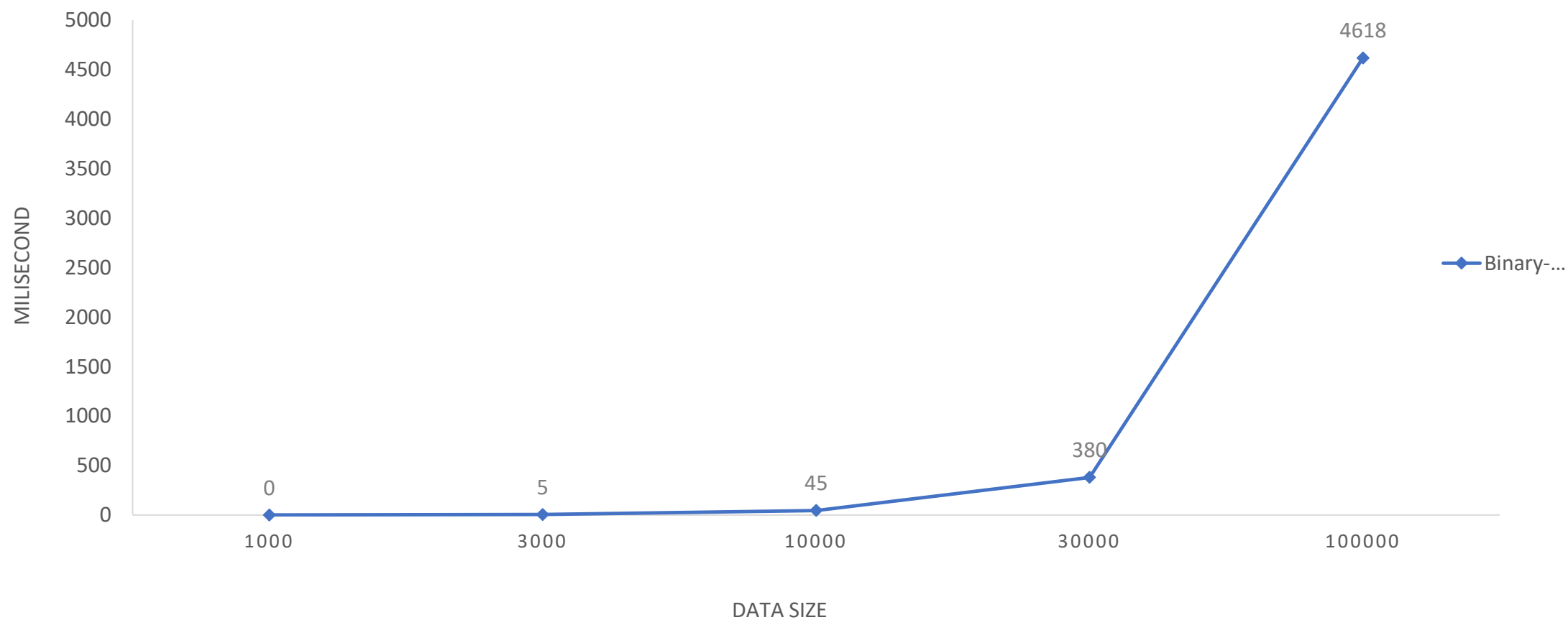
ALGORITHM REVIEW

The binary-insertion sort performs $\lceil \log_2 n \rceil$ comparisons in the worst case, which is $O(n \log n)$. The algorithm as a whole still has a running time of $O(n^2)$ on average because of the series of swaps required for each insertion.

Case	Time complexity
Best	$O(n \log n)$
Average	$O(n^2)$
Worst	$O(n^2)$

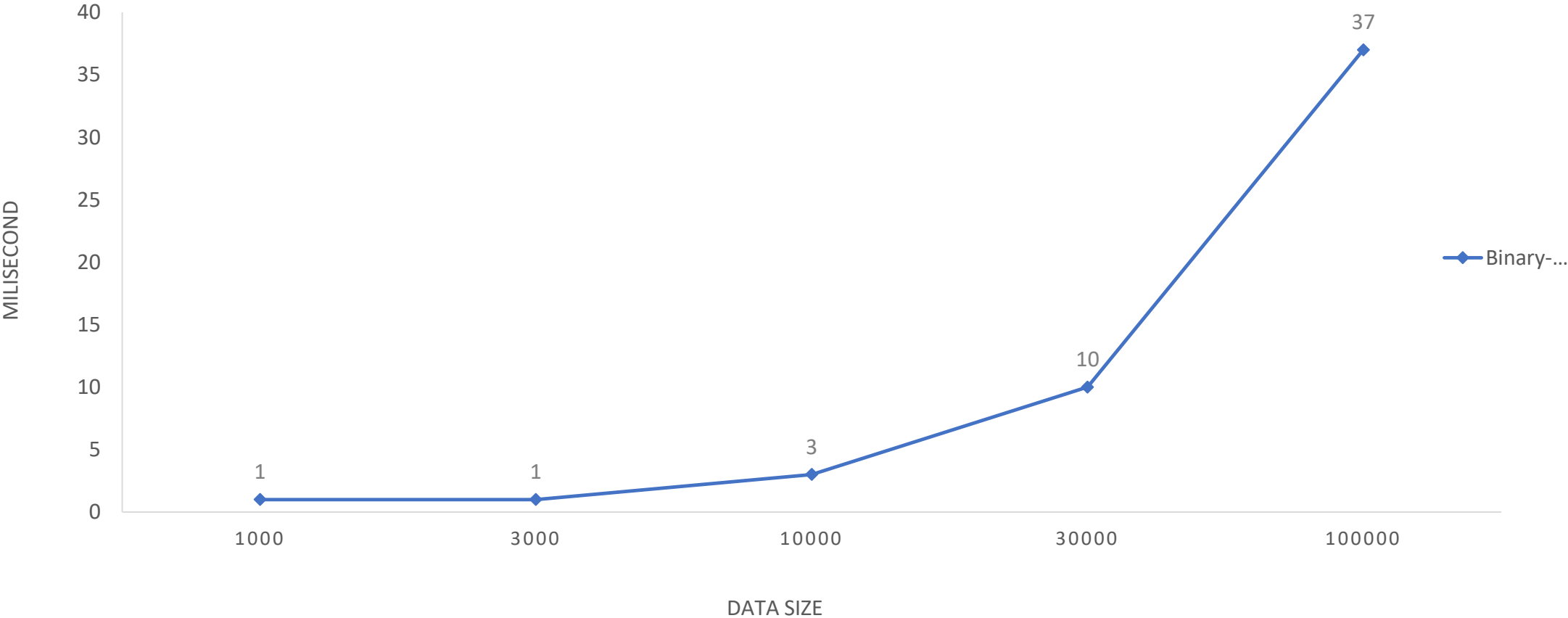
GRAPH

RANDOM DATA TYPE



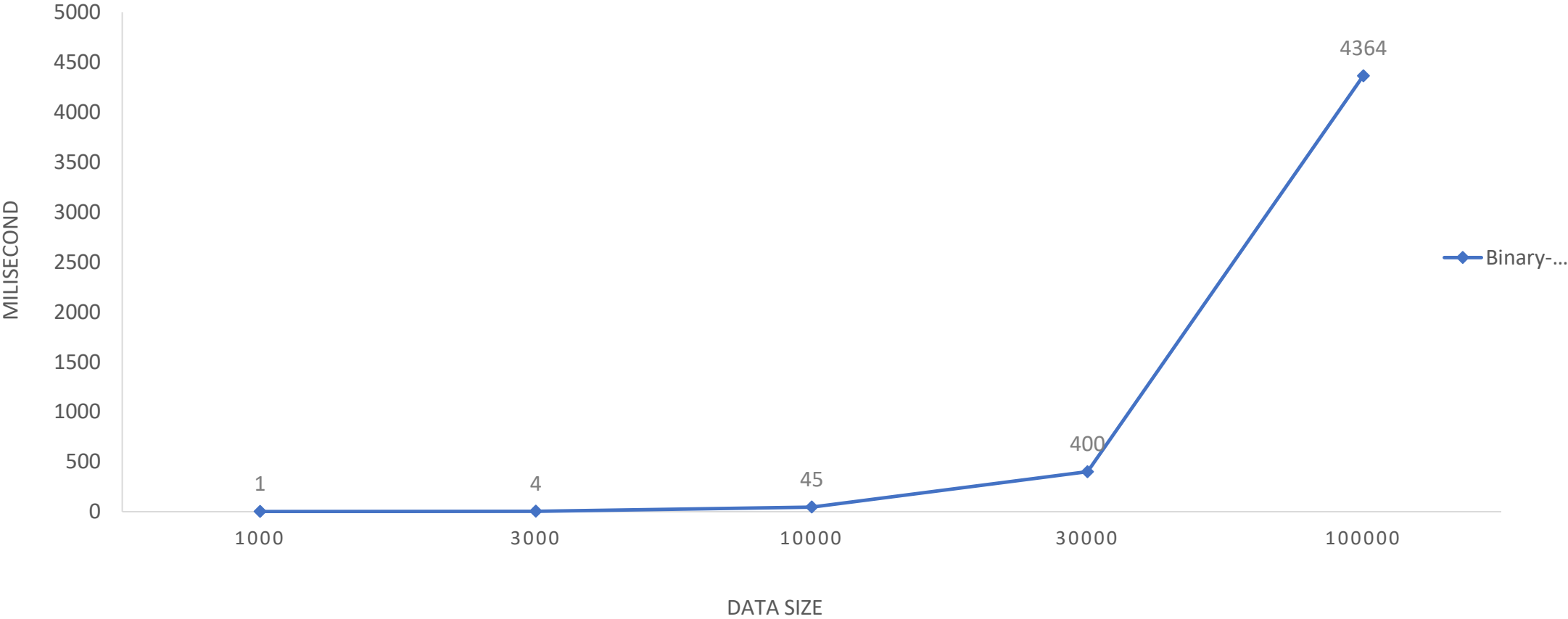
GRAPH

SORTED DATA TYPE



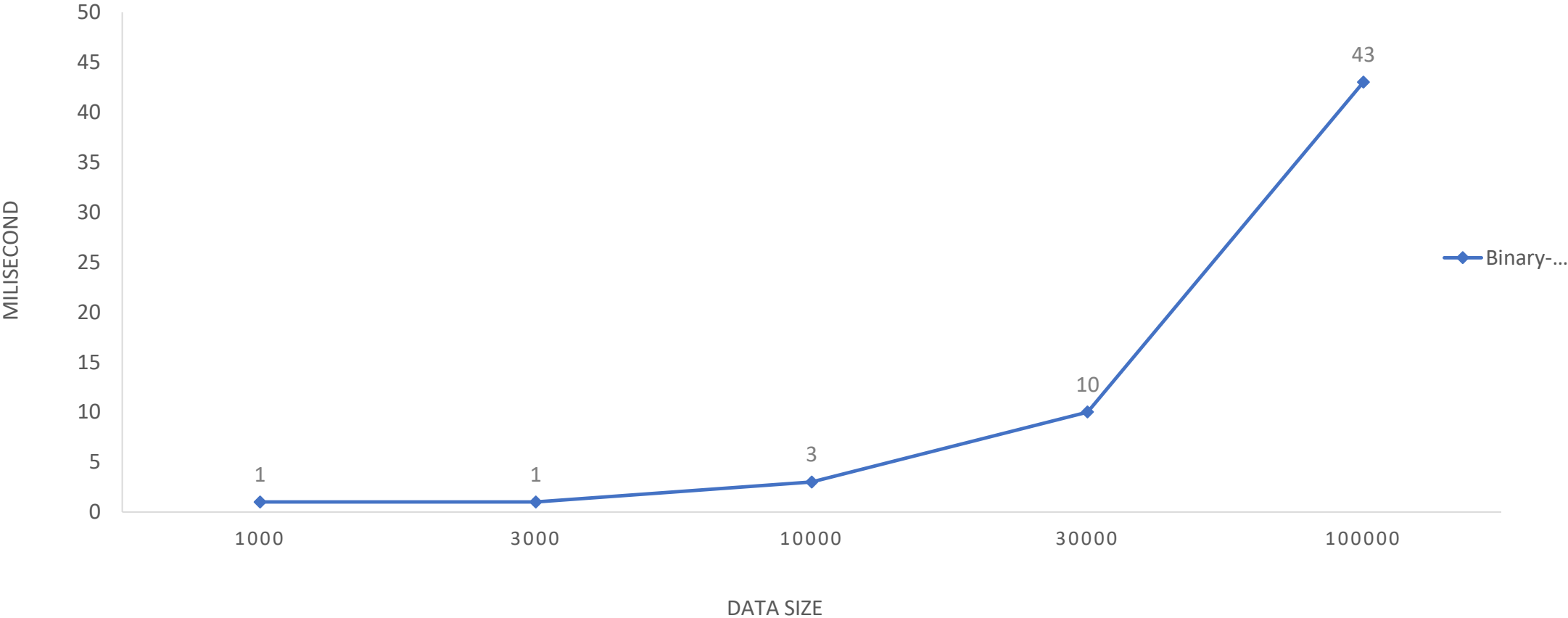
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE





FLASH SORT

IDEA – ALGORITHM – ALGORITHMS REVIEW

IDEA

The basic idea behind *flashsort* is that in a data set with a known distribution, it is easy to immediately estimate where an element should be placed after sorting when the range of the set is known. For example, if given a uniform data set where the minimum is 1 and the maximum is 100 and 50 is an element of the set, it's reasonable to guess that 50 would be near the middle of the set after it is sorted. This approximate location is called a class. If numbered 1 to m , the class of an item A_i is the quantile, computed as:

$$K(A_i) = 1 + \text{int}((m - 1) \frac{A_i - A_{\min}}{A_{\max} - A_{\min}})$$

where A is the input set. The range covered by every class is equal, except the last class which includes only the maximum(s). The classification ensures that every element in a class is greater than any element in a lower class. This partially orders the data and reduces the number of inversions. Insertion sort is then applied to the classified set. As long as the data is uniformly distributed, class sizes will be consistent and insertion sort will be computationally efficient

ALGORITHM

DATA STRUCTURE AND ALGORITHMS

CODE REFERENCE

Flash Sort

```
void FlashSort(int arr[], int n) {
    if (n == 0) return;
    //20% of the number of elements or 0.2n classes will
    //be used to distribute the input data set into
    //there must be at least 2 classes (hence the addition)
    int m = (int)((0.2 * n) + 2);
    //-----CLASS FORMATION-----
    //O(n)
    //compute the max and min values of the input data
    int min, max, maxIndex;
    min = max = arr[0];
    maxIndex = 0;

    for (int i = 1; i < n - 1; i += 2){
        int small;
        int big;
        int bigIndex;

        if (arr[i] < arr[i + 1]){
            small = arr[i];
            big = arr[i + 1];
            bigIndex = i + 1;
        } else{
            big = arr[i];
            bigIndex = i;
            small = arr[i + 1];
        }
        if (big > max){
            max = big;
            maxIndex = bigIndex;
        }
        if (small < min){
            min = small;
        }
    }
}
```

```
//do the last element
if (arr[n - 1] < min){
    min = arr[n - 1];
}
else if (arr[n - 1] > max){
    max = arr[n - 1];
    maxIndex = n - 1;
}
if (max == min){
    //all the elements are the same
    return;
}
//dynamically allocate the storage for L
//note that L is in the range 1...m (hence
//the extra 1)
int* L = new int[m + 1]();
//O(n)
//use the function  $K(A(i)) = 1 + \text{INT}((m-1)(A(i)-A_{\min})/(A_{\max}-A_{\min}))$ 
//to classify each  $A(i)$  into a number from 1...m
//(note that this is mainly just a percentage calculation)
//and then store a count of each distinct class  $K$  in  $L(K)$ 
//For instance, if there are 22  $A(i)$  values that fall into class
// $K == 5$  then the count in  $L(5)$  would be 22
//IMPORTANT: note that the class  $K == m$  only has elements equal to  $A_{\max}$ 
//precomputed constant
double c = (m - 1.0) / (max - min);
int K;
for (int h = 0; h < n; h++){
    //classify the  $A(i)$  value
    K = ((int)((arr[h] - min) * c)) + 1;
    //assert: K is in the range 1...m
    //add one to the count for this class
    L[K] += 1;
}
//O(m)
//sum over each  $L(i)$  such that each  $L(i)$  contains
//the number of  $A(i)$  values that are in the  $i$ th
//class or lower (see counting sort for more details)
for (K = 2; K <= m; K++){
    L[K] = L[K] + L[K - 1];
}
}
```

Flash Sort

```
//-----PERMUTATION-----

//swap the max value with the first value in the array
int temp = arr[maxIndex];
arr[maxIndex] = arr[0];
arr[0] = temp;

//Except when being iterated upwards,
//j always points to the first A(i) that starts
//a new class boundary && that class hasn't yet
//had all of its elements moved inside its borders;

//This is called a cycle leader since you know
//that you can begin permuting again here. You know
//this because it is the lowest index of the class
//and as such A(j) must be out of place or else all
//the elements of this class have already been placed
//within the borders of the this class (which means
//j wouldn't be pointing to this A(i) in the first place)
int j = 0;

//K is the class of an A(i) value. It is always in the range 1..m
K = m;

//the number of elements that have been moved
//into their correct class
int numMoves = 0;

//O(n)
//permute elements into their correct class; each
//time the class that j is pointing to fills up
//then iterate j to the next cycle leader
//
//do not use the n - 1 optimization because that last element
//will not have its count decreased (this causes trouble with
//determining the correct classSize in the last step)
```

```
while (numMoves < n)
{
    //if j does not point to the beginning of a class
    //that has at least 1 element still needing to be
    //moved to within the borders of the class then iterate
    //j upward until such a class is found (such a class
    //must exist). In other words, find the next cycle leader
    while (j >= L[K])
    {
        j++;

        K = ((int)((arr[j] - min) * c)) + 1;
    }
    //evicted always holds the value of an element whose location
    //in the array is free to be written into //aka FLASH
    int evicted = arr[j];

    //while j continues to meet the condition that it is
    //pointing to the start of a class that has at least one
    //element still outside its borders (the class isn't full)
    while (j < L[K])
    {
        //compute the class of the evicted value
        K = ((int)((evicted - min) * c)) + 1;

        //get a location that is inside the evicted
        //element's class boundaries
        int location = L[K] - 1;

        //swap the value currently residing at the new
        //location with the evicted value
        int temp = arr[location];
        arr[location] = evicted;
        evicted = temp;

        //decrease the count for this class
        //see counting sort for why this is done
        L[K] -= 1;

        //another element was moved
        numMoves++;
    }
}
```

Flash Sort

```
int threshold = (int)(1.25 * ((n / m) + 1));
const int minElements = 30;

for (K = m - 1; K >= 1; K--)
{
    int classSize = L[K + 1] - L[K];

    if (classSize > threshold && classSize > minElements)
    {
        doFlashSort(&arr[L[K]], classSize);
    }
    else
    {
        if (classSize > 1)
        {
            insertionSort(&arr[L[K]], classSize);
        }
    }
}

delete[] L;
}
```

ALGORITHM REVIEW

Flashsort avoids the overhead needed to store classes in the very similar bucketsort. For $m=0.1n$ with uniform random data, flashsort is faster than heapsort for all n and faster than quicksort for $n>80$. It becomes about as twice as fast as quicksort at $n=10000$.

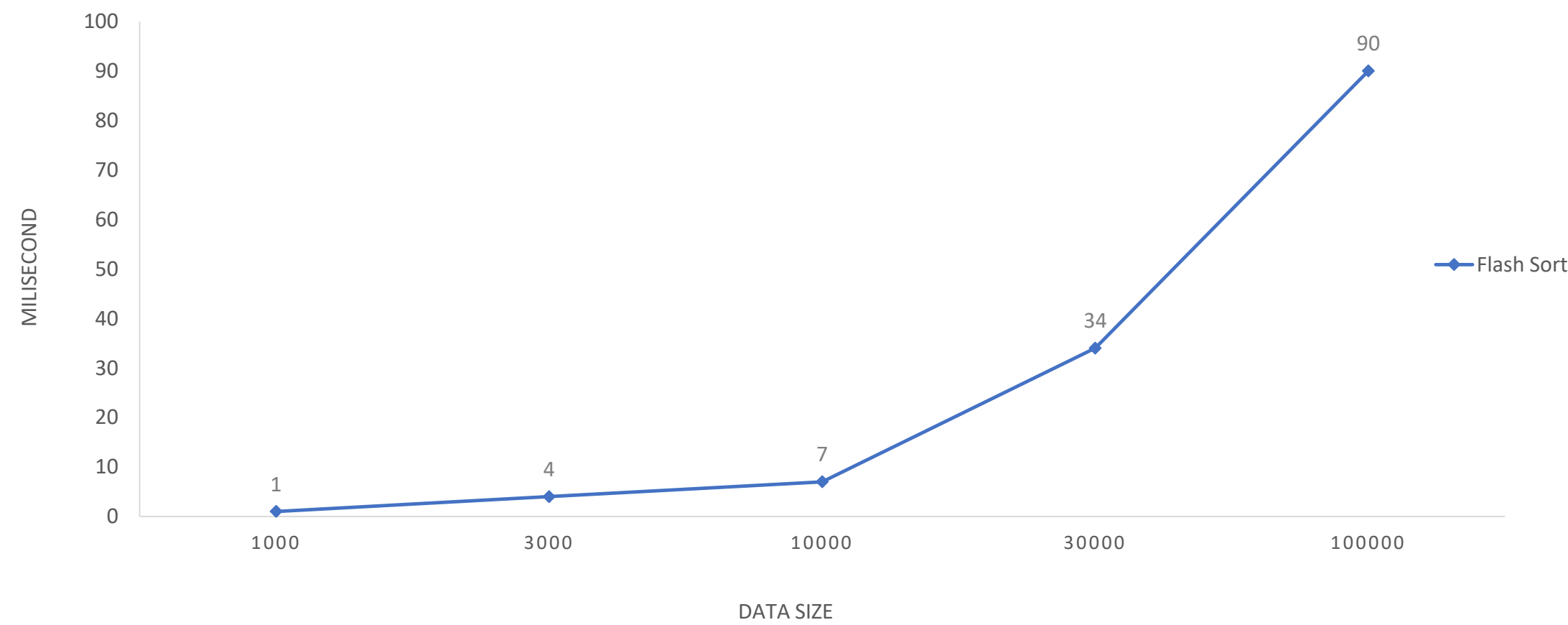
Due to the in situ permutation that flashsort performs in its classification process, flashsort is not stable. If stability is required, it is possible to use a second, temporary, array so elements can be classified sequentially. However, in this case, the algorithm will require $O(n)$ space.

Time complexity

$O(n)$

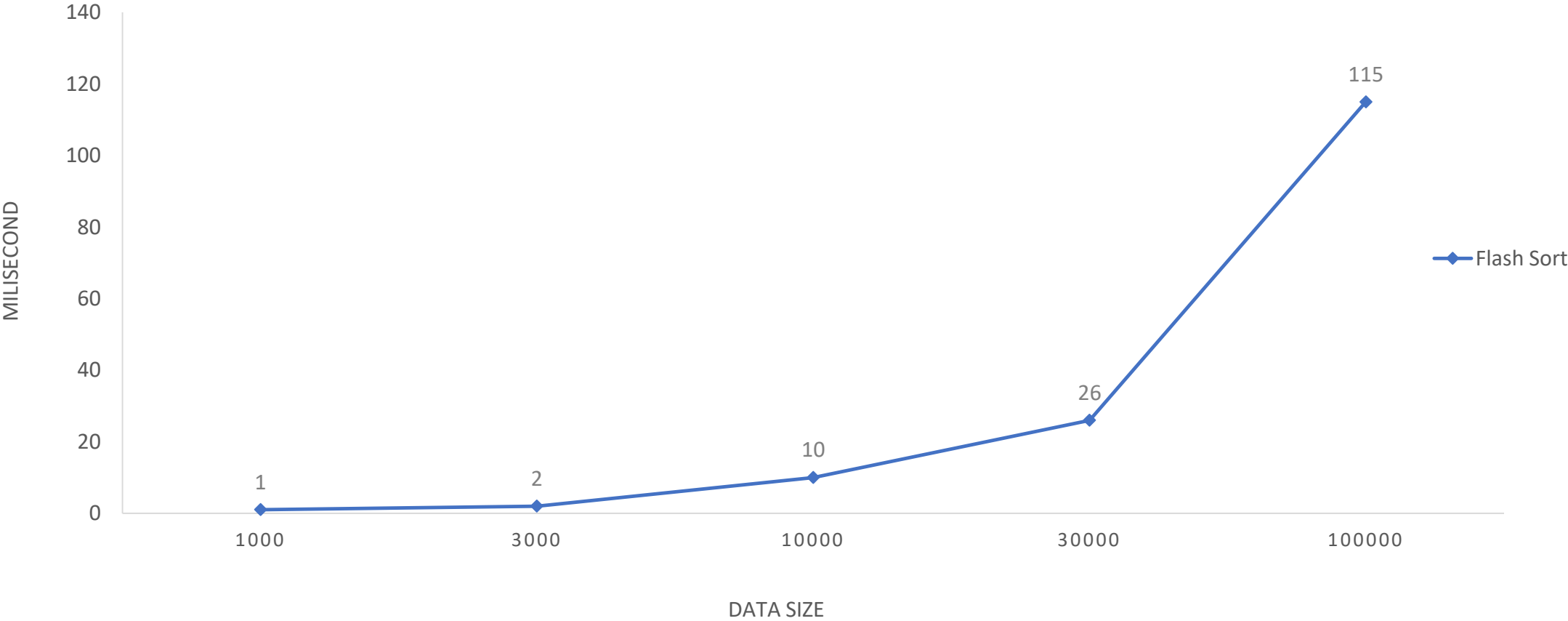
GRAPH

RANDOM DATA TYPE



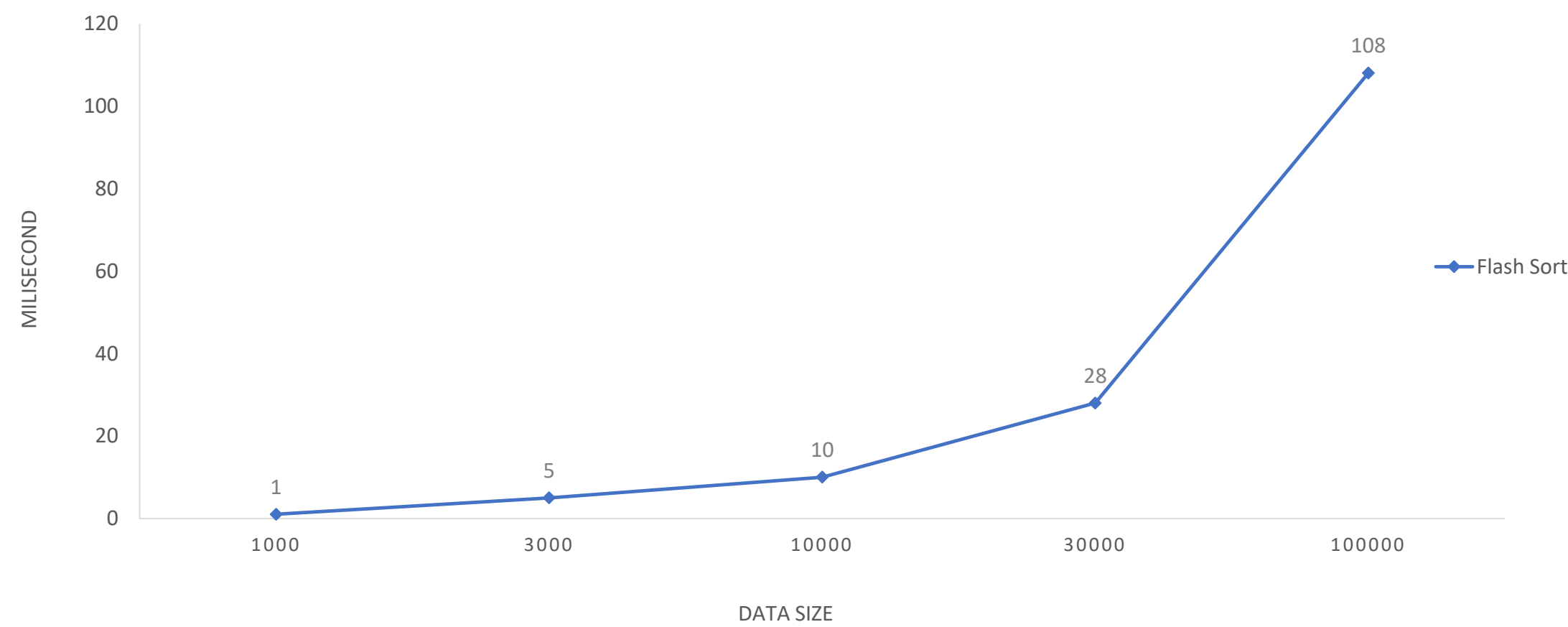
GRAPH

SORTED DATA TYPE



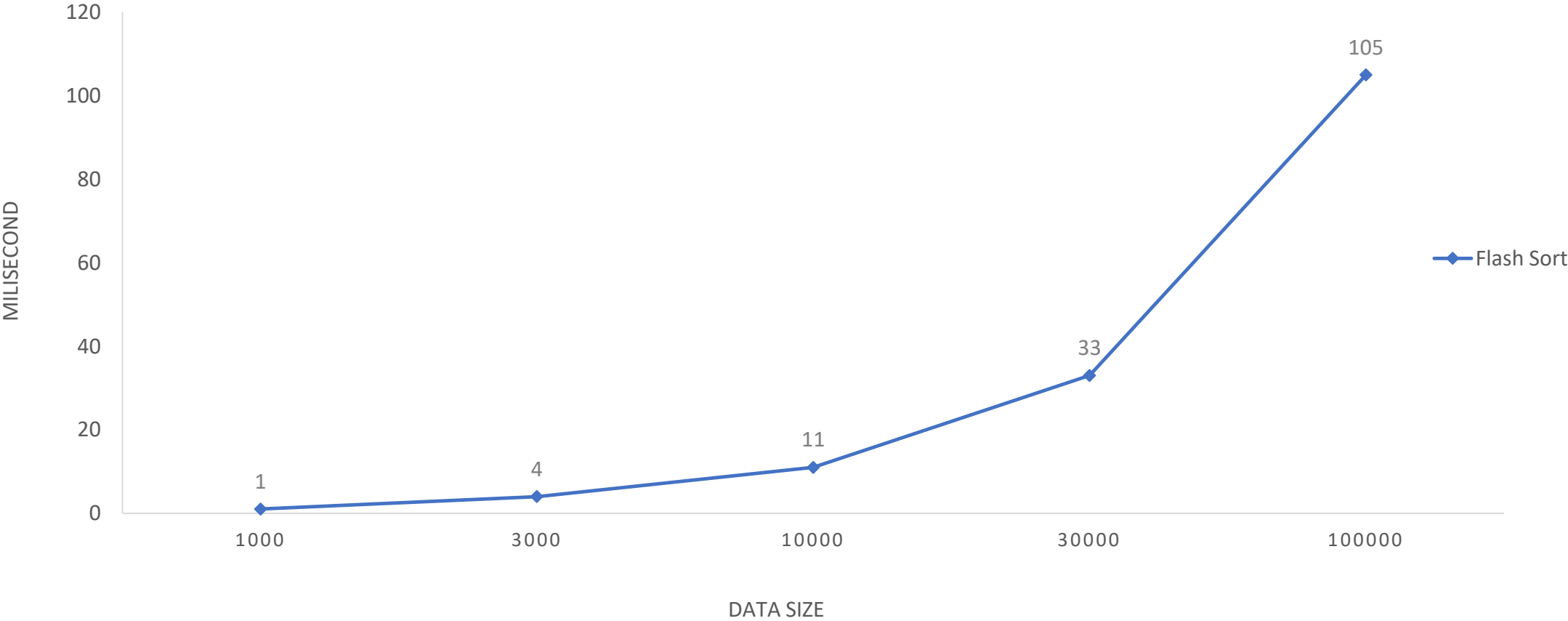
GRAPH

REVERSE DATA TYPE



GRAPH

NEARLY SORTED DATA TYPE

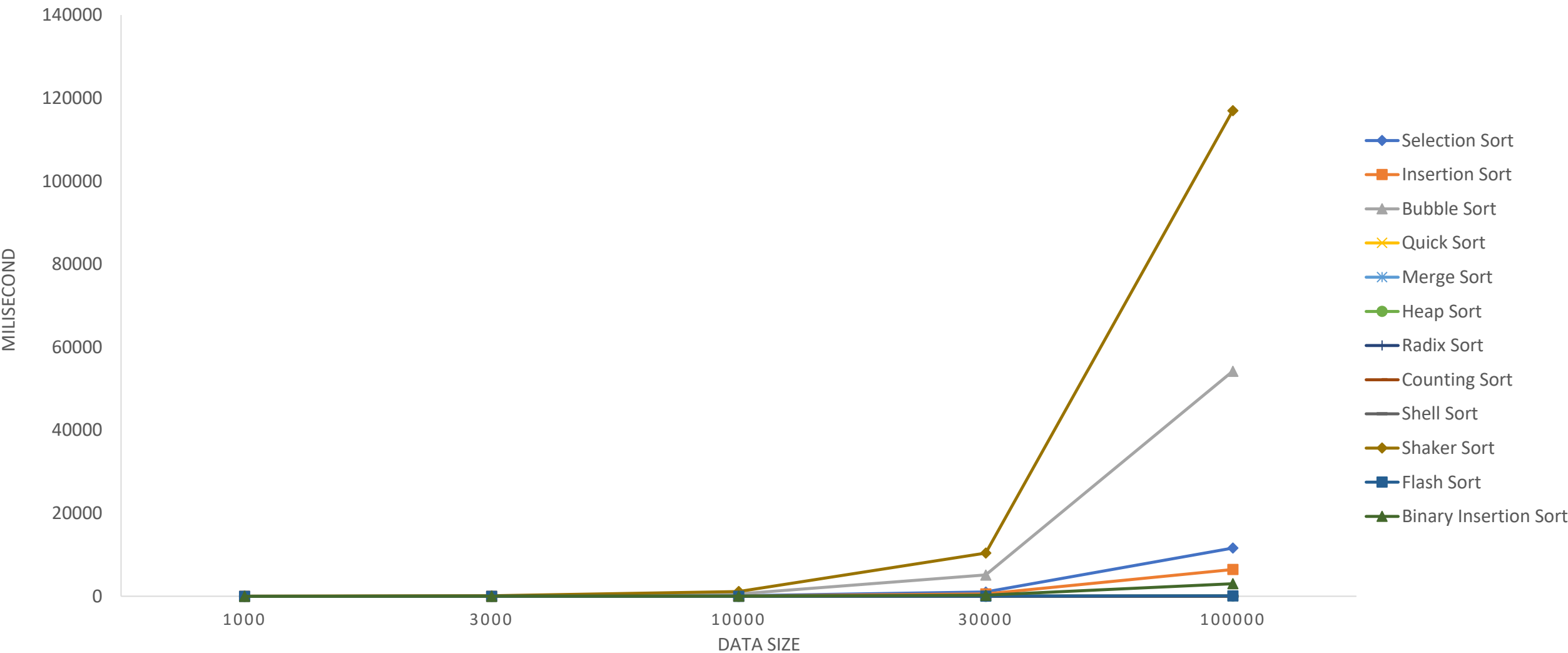




SUMMARIZE

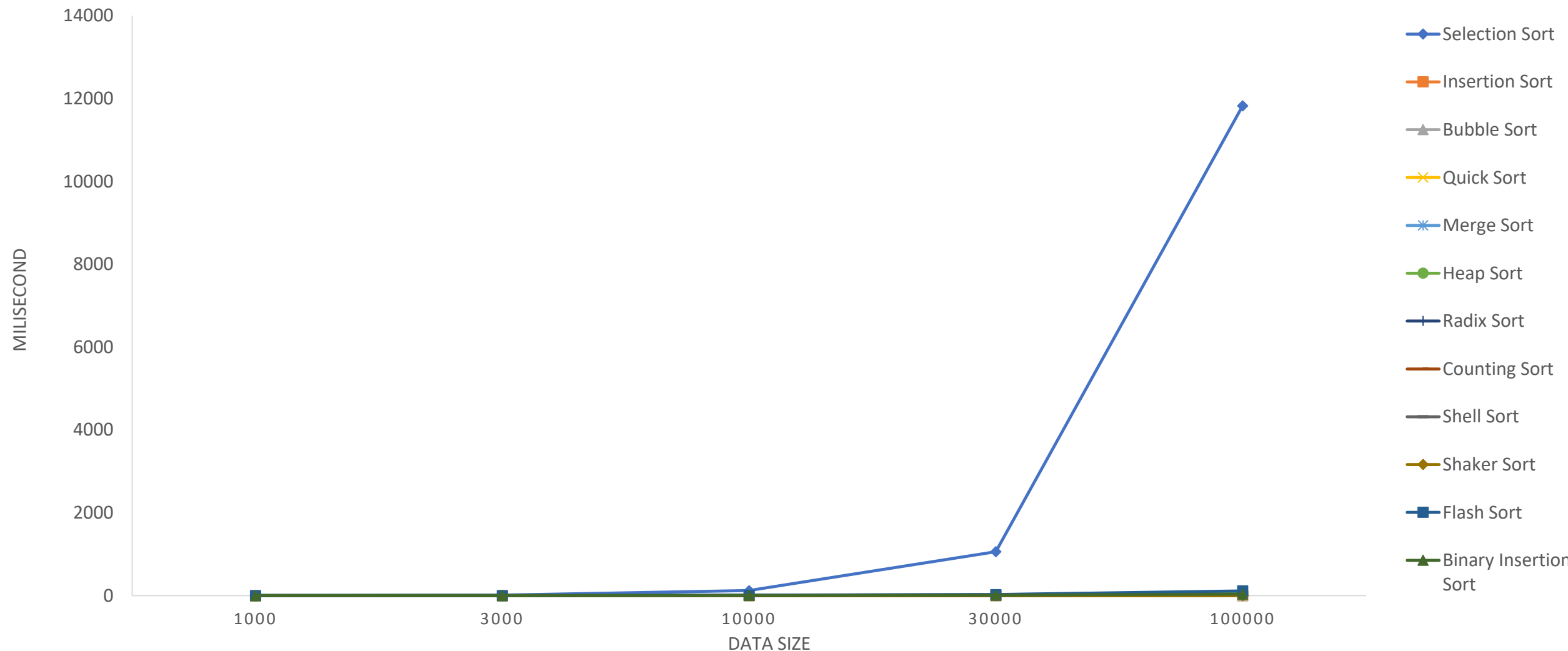
GRAPH

RANDOM DATA TYPE



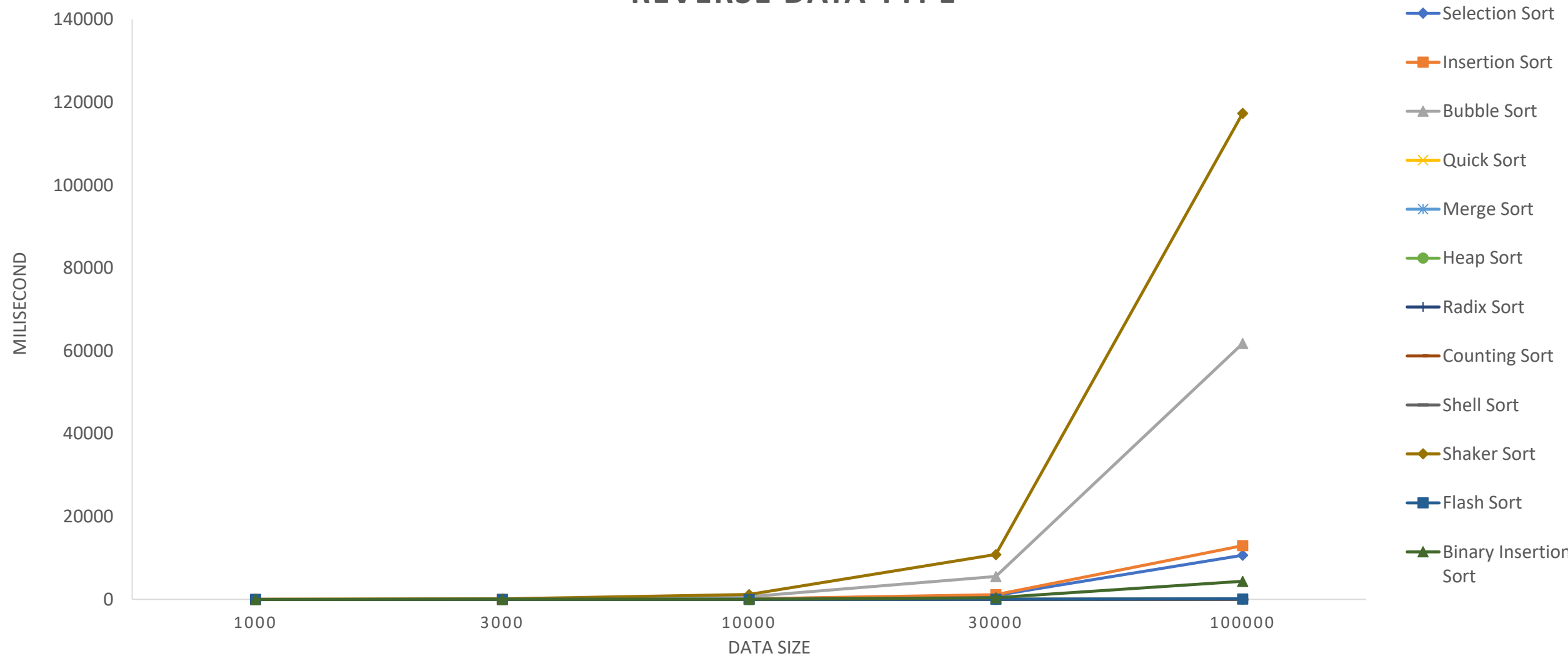
GRAPH

SORTED DATA TYPE



GRAPH

REVERSE DATA TYPE



GRAPH

[illegible]