Project Report

Rohan Girish (2021101042), Sarthak Chittawar (2021111010)

Problem Statement:

Our Project is to run a comparative analysis of library sort with other sorting algorithms:

1) Insertion sort
2) Quick sort
3) Merge sort

and to see how it performs in real scenarios with the following types of input arrays:

1) Almost sorted arrays
2) Reverse sorted arrays
3) Randomly generated arrays.

We have also tested for different gap sizes, from 1 to 10 (we didn't go higher due to memory limits and the fact that for small sized arrays run times were increasing after epsilon 2 or 3, and using too many epsilon values makes it difficult to graph).

We have also created a new preprocessing step for library sort, which we have described later, and noticed there is a decrease in the run time as well as memory (it was built for reducing memory usage, but we noticed it reduces run times as well).

The Library Sort Algorithm:

The algorithm we implemented is taken from the Bader paper.

Initializations:

```
int tot = ceil_log2(size);
int *newL = (int *)malloc(sizeof(int) * tot * (1 + epsilon));
int curr_size = 1 + epsilon;
 for (int i = 0; i < tot * (1 + epsilon); i++)
{
    newL[i] = -1;
}
```

ceil_log2(size) gives us the power of 2 immediately greater than the size of our array (since we keep multiplying by 2 till the whole array fits), and we multiply with 1+ε in order to account for gaps (as mentioned in the paper, at any given point our array size is 2^i(1+ε), starting from i=0). We initialize our array to have only -1 as an element (to signify gaps). We have used only positive elements in our array to make it easier to signify gaps using -1, as positive/negative shouldn't affect relative run times for different algorithms.

The main section:

```
for (int i = 0; i < size; i++)
   {
       if (i == curr_size / (1 + epsilon))
       {
           curr_size *= 2;
           // if(size==524288){
           //     cout << curr_size << endl;
           // }
           rebalance(newL, curr_size, epsilon);
       }

       int curr = L[i];
       // cout << i << " ";
       int start = binSearch(newL, 0, curr_size - 1, curr);
```

```
        int j = start;
        if (j == -1)
        {
            // cout << i << " ";
            newL[i] = curr;
            continue;
        }
        // cout << start << " ";
        for (j; newL[j] != -1; j++)
        {
            // cout << curr << " ";
            if (curr < newL[j])
            {
                int temp = curr;
                curr = newL[j];
                newL[j] = temp;
            }
        }
        newL[j] = curr;
        // check_sorted(newL,curr_size);
    }
```

In the first if statement, if the currently added number of elements is equal to our current power of 2, we multiply the size of the array by 2 and rebalance the array (rebalance function shown in the next part). The part after the if statement - we binary search (explained after the rebalance function) for the element directly greater than the element we are inserting, add it in that place and shift every successive element till we reach a gap.

The Rebalance function:

```
void rebalance(int *L, int size, int epsilon)
{
    int *newL = (int *)malloc(sizeof(int) * size);
    int curr = 0;
    for (int i = 0; i < (size / 2); i++)
    {
        // cout << L[i] << " ";
```

```
        if (L[i] == -1)
        {
            continue;
        }
        for (int j = 0; j < epsilon; j++)
        {
            newL[curr++] = -1;
        }
        newL[curr++] = L[i];
    }
    for (int i = 0; i < size; i++)
    {
        if (curr == i)
        {
            newL[curr++] = -1;
        }
        // cout << newL[i] << " ";
        L[i] = newL[i];
    }
    // cout << endl;
}
```

We just redistribute the elements of the array, leaving ε gaps between 2 consecutive elements, preserving the order (since the order is already sorted).

The Binary Search function:

```
int binSearch(int *L, int low, int high, int val)
{

    if (high <= low)
    {
        if (low < 0)
        {
            return -1;
        }
        if (L[low] == -1)
        {
```

```
            return low;
        }
        if (L[low] < val)
        {
            return low + 1;
        }
        else
        {
            return low;
        }
    }

    int mid1 = (low + high) / 2;
    int mid2 = (low + high) / 2;
    while (L[mid1] == -1 and mid1 >= low)
    {
        mid1--;
    }
    while (L[mid2] == -1 and mid2 <= high)
    {
        mid2++;
    }
    if (L[mid1] == -1 && L[mid2] == -1)
    {
        return -1;
    }

    if (mid1 < low && mid2 > high)
    {
        mid1 = (low + high) / 2;
        return low;
    }

    else if (mid1 >= low && mid2 > high)
    {
        if (L[mid1] == val)
        {
            return mid1;
        }
        else if (L[mid1] > val)
```

```
        {
            return binSearch(L, low, mid1 - 1, val);
        }
        else
        {
            return binSearch(L, mid1 + 1, high, val);
        }
    }

    else if (mid1 < low && mid2 <= high)
    {
        if (L[mid2] == val)
        {
            return mid2;
        }
        else if (L[mid2] < val)
        {
            return binSearch(L, mid2 + 1, high, val);
        }
        else
        {
            return binSearch(L, low, mid2 - 1, val);
        }
    }
    else
    {
        if (L[mid1] == val)
        {
            return mid1;
        }
        else if (L[mid2] == val)
        {
            return mid2;
        }
        else if (L[mid1] > val)
        {
            return binSearch(L, low, mid1 - 1, val);
        }
        else if (L[mid2] < val)
        {
```

```
            return binSearch(L, mid2 + 1, high, val);
        }
    }

    return -1;
}
```

This is a modified Binary Search algorithm, which accounts for gaps present between elements. (It does this by iterating backwards and forwards from a gap till the specified left/right, and stopping if it encounters an element and using the normal binary search algorithm if it does. If it doesn't, it returns the midpoint of the subsection we are searching in, and we insert our element there).

The final section:

```
    int curr = 0;
    // cout << "Here" << endl;
    for (int i = 0; i < size; i++)
    {
        // cout << newL[curr] << " ";

        if (newL[curr++] == -1)
        {
            i--;
            continue;
        }
        L[i] = newL[curr - 1];
    }
    // cout << endl;
    free(newL);
```

This just moves the elements back into our original array in sorted order (removing gaps), and frees our temporary array.

Our Optimization:

As was stated before, our temporary array has a size of ceil_log2(size)*(1+ε). However, this ceil_log2(size) is needless and the same effect can be obtained in O(N*(1+ε)).

We have done this by converting N into sums of powers of 2 (using the binary representation of N), and running library sort on each of these individually and finally merging. This brings down the memory usage by (ceil_log(size)-N)*(1+ε), but adds an overhead time of O(N) while merging.However, even with the overhead time, the actual run time may decrease due to the reduction in number of unnecessary gaps in the temporary array, which makes its iteration easier and hence reduces the run time of the actual Sorting Section.
(We also noticed that Library Sort usually runs much quicker for powers of 2).

Structs used:

```
typedef struct Node
{
   int size;
   int curr;
   struct Node *next;
   struct Node *prev;
} node;

typedef struct LL
{
   node *head;
   node *tail;
   int size;
} ll;
node *createNode(int prev, int i, ll *sizes)
{
   node *newNode = (node *)malloc(sizeof(node));
   newNode->prev = sizes->tail;
   newNode->next = NULL;
```

```
    newNode->curr = prev;
    newNode->size = prev + (1 << i);
    // cout << newNode->curr << " " << newNode->size << endl;
    if (newNode->prev != NULL)
    {
        newNode->prev->next = newNode;
        sizes->tail = newNode;
    }
    else
    {
        sizes->head = newNode;
        sizes->tail = newNode;
    }

    return newNode;
}

void freeNode(node *myNode, ll *sizes)
{
    if (myNode->prev == NULL)
    {
        sizes->head = myNode->next;
    }
    else
    {
        myNode->prev->next = myNode->next;
    }
    if (myNode->next == NULL)
    {
        sizes->tail = myNode->prev;
    }
    else
    {
        myNode->next->prev = myNode->prev;
    }

    free(myNode);
}
```

We use a linked list to represent our subarrays (reduces merging run time while using slightly higher constant memory, as if we already inserted all elements of a subarray we don't need to check it repeatedly in every iteration and we can remove it from the linked list in O(1) time).

The preprocessing function:

```
void preProcessingLibSort(int *L, int size, int epsilon)
{
   int curr = size;
   int i = 0;
   int num = 0;
   while (curr > 0)
   {
       num += curr & 1;
       curr >>= 1;
   }

   // cout << num << endl;

   curr = size;
   // int sizes[num+1];
   int prev = 0;
   ll *sizes = (ll *)malloc(sizeof(ll));
   sizes->head = NULL;
   sizes->tail = sizes->head;
   sizes->size = num;
   // sizes[0]=0;
   int curr_index = 1;
   while (curr > 0)
   {
       if (curr & 1)
       {
           // sizes[curr_index] = sizes[curr_index-1] + (1<<i);
           createNode(prev, i, sizes);
           // cout << sizes[curr_index] << " " << sizes[curr_index-1] << "
" << (1<<i) << endl;
           curr_index++;
```

```
            // cout << (1<<i) << endl;
            normLibSort(L + prev, (1 << i), epsilon);
            // cout << "Done" << endl;
            prev += (1 << i);
        }
        curr >>= 1;
        i++;
    }

    libMerge(L, size, sizes);
}
```

As explained before, it takes sums of powers of 2 and sorts them individually before merging.

Merging:

```
void libMerge(int *L, int size, ll *sizes)
{
    int *newL = (int *)malloc(sizeof(int) * size);
    // ll* merging = (ll*)malloc(sizeof(ll));
    // merging->head = NULL;

    for (int i = 0; i < size; i++)
    {
        node *myNode = sizes->head;
        int curr = L[sizes->head->curr];
        node *currNode = sizes->head->next;
        while (currNode != NULL)
        {
            if (L[currNode->curr] < curr)
            {
                curr = L[currNode->curr];
                myNode = currNode;
            }
            currNode = currNode->next;
        }
        newL[i] = curr;
        myNode->curr++;
        if (myNode->curr == myNode->size)
        {
```

```
            freeNode(myNode, sizes);
        }
        if (sizes->head == NULL)
        {
            break;
        }
    }

    free(sizes);

    for (int i = 0; i < size; i++)
    {
        L[i] = newL[i];
    }
    free(newL);
}
```

As explained, keep checking for all subarrays and when it is fully inserted, free the node representing it.

Mathematically, as seen from our implementation, we are reducing the memory usage by (ceil_log(size)-size)*(1+ε)

Further improvements and optimizations:

1) Check if the segment is already sorted or not. If it is, no need to call library sort again.
2) First divide the array into sorted and unsorted segments, and then do the optimized library sort on the unsorted segments before merging.
3) Run the merges in parallel so that the run time of each merge is bounded by the slowest merge (especially helpful since we know we'll have 32 divisions in the worst case (number of bits in an integer)).

Our Graphs:

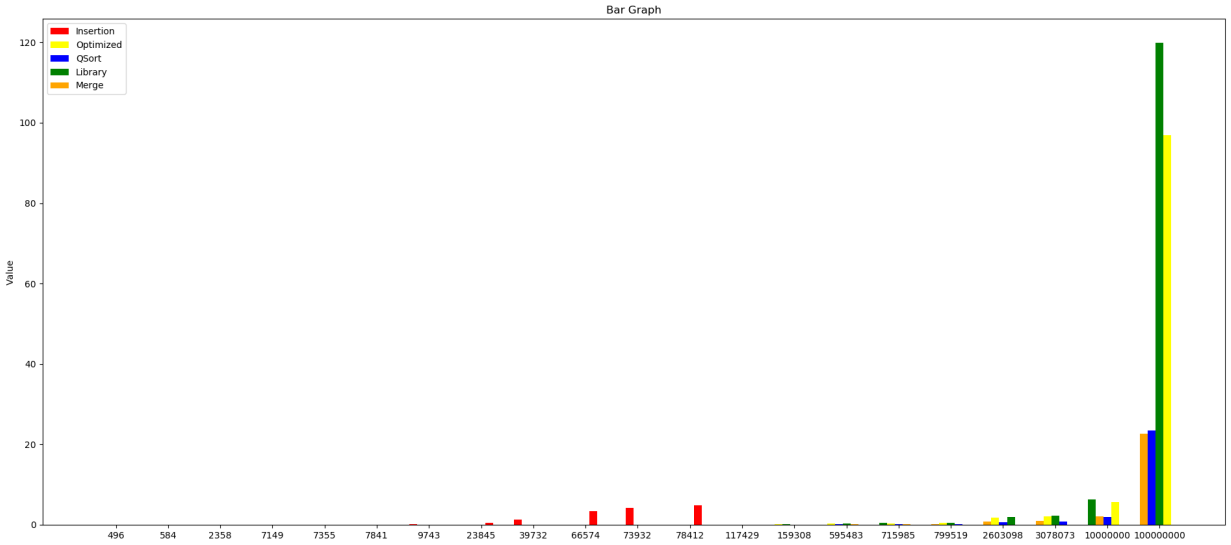
Note: For every graph, the y axis denotes the run time (in seconds) and the value in the x axis is the array size. We used random array sizes of certain orders, to remove the bias of running only on powers of 10. Please note the scaling for each of the graphs, as scaling changes based on max value used in the graph to provide better visibility.

Based on Run Times:

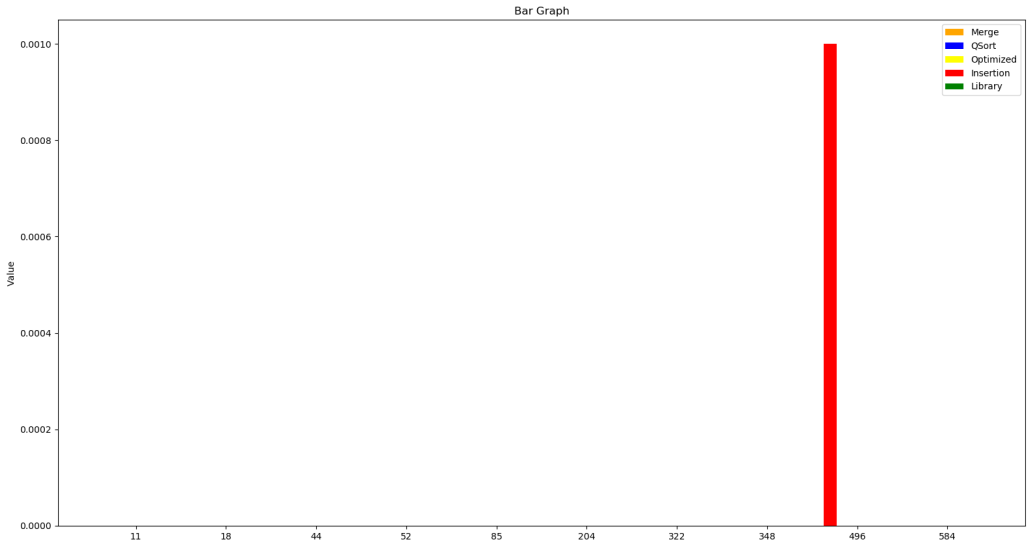
For library sort and optimized library sort, we have plotted the graphs taking the best performing epsilon value from 1 to 10. For each size, we run either 10 or 100 different randomly generated arrays (based on time taken, as running it for too long even on ssh was not feasible due to the effect keeping our laptops plugged in for that long would have on our battery health), after which we sort or reverse sort or do nothing based on the requirement, and then we take the average of those run times and plot for that size.

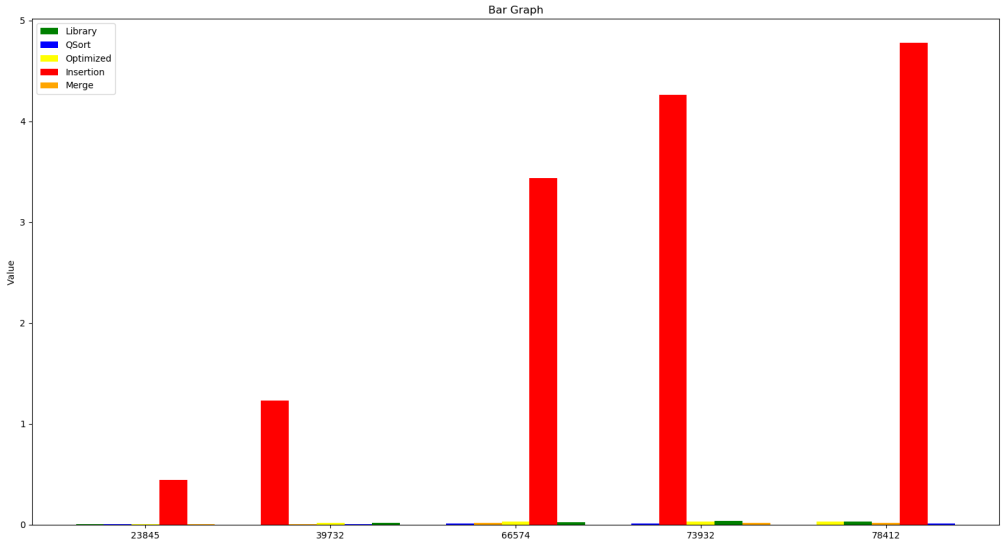
1) **Fully Randomized:**

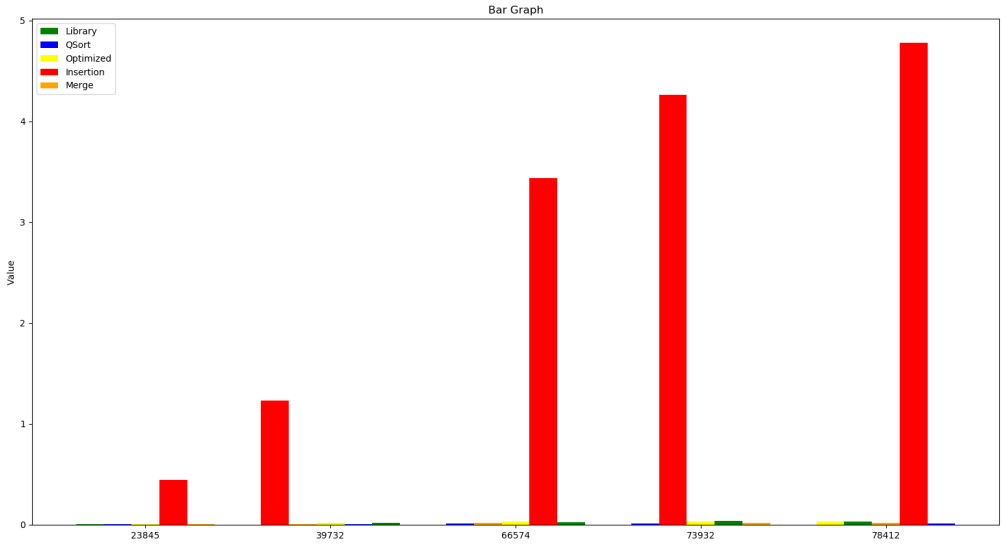
Overall graph for all sizes



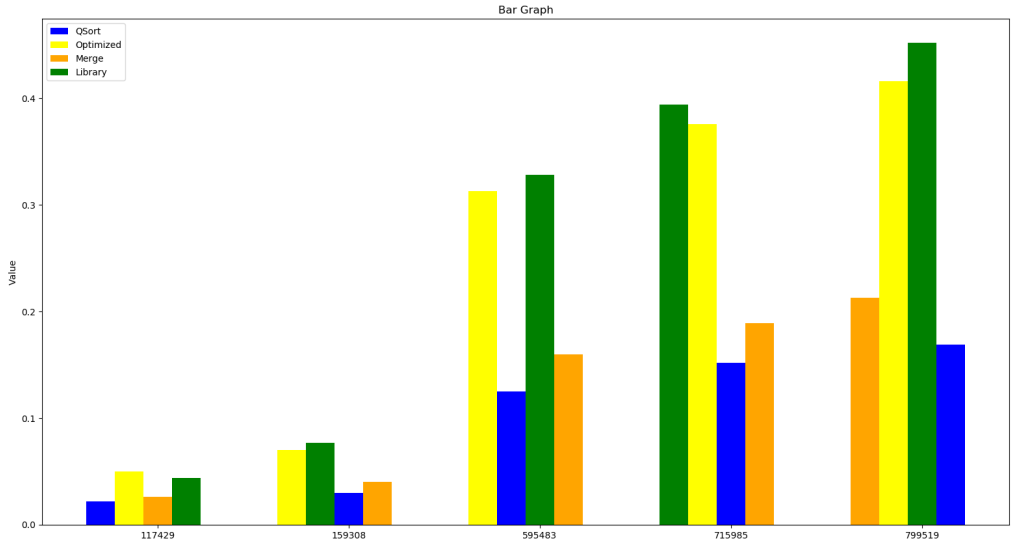
Graphs taken a few at a time for better visibility of bars for comparison







At this point, we removed insertion sort due to the huge run time difference between it and the rest of the sorts, reducing visibility and comparability between library sort and library sort with our optimization, which is our secondary aim.

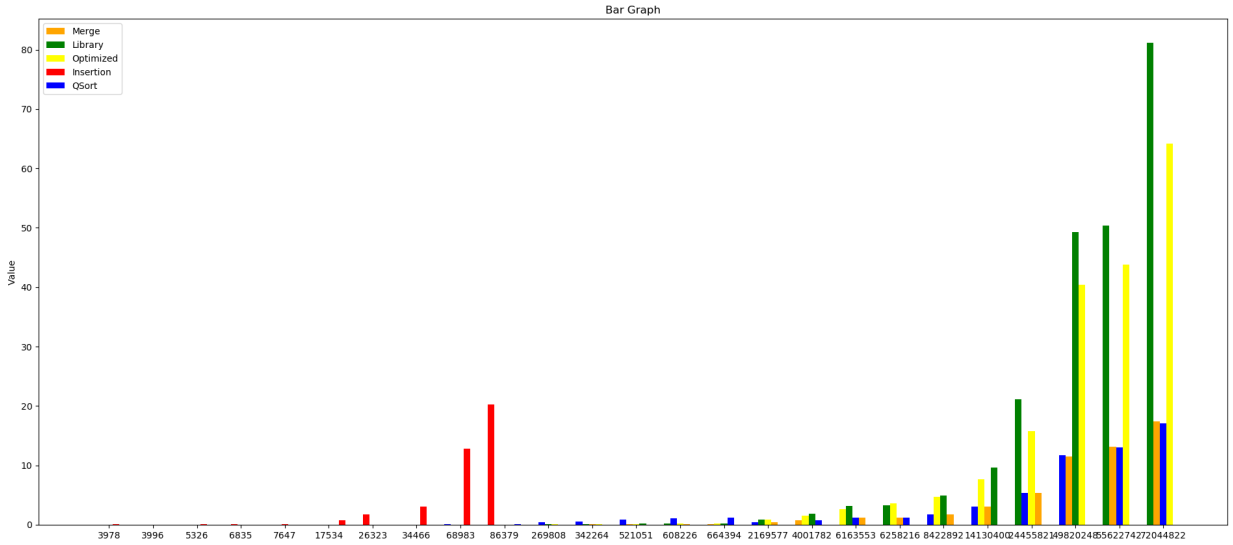


Observations:

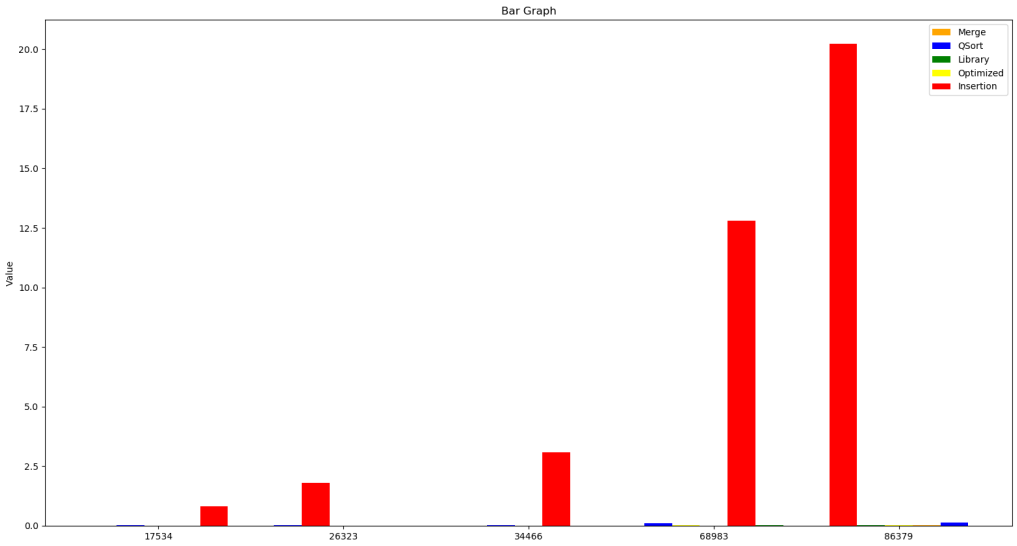
- For all array sizes, Library Sort (& Optimized) takes much less time than insertion sort.
- QSort and Merge Sort are, however, quicker than the other two for all array sizes.

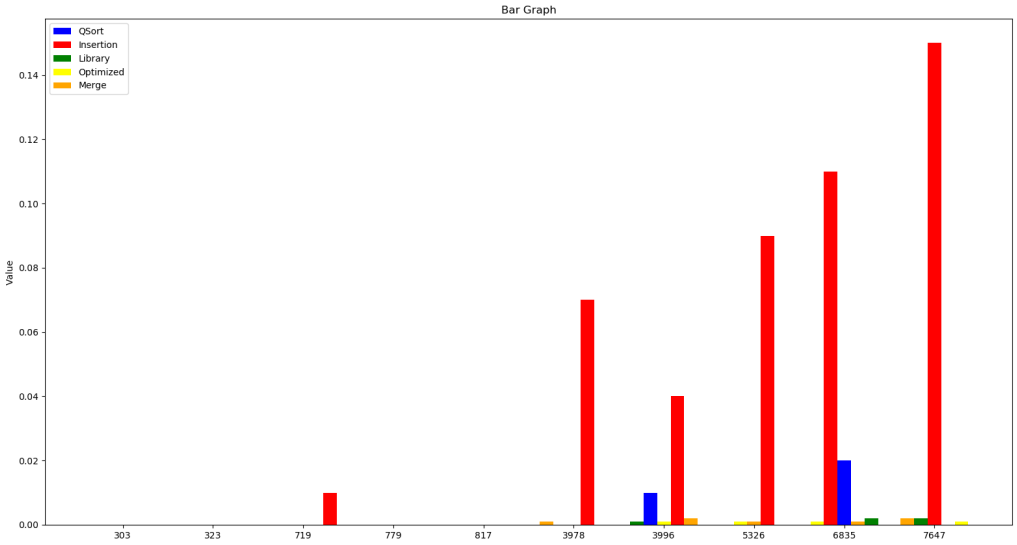
2) Partially Sorted: (We used half sorted half random)

Overall graph for all sizes

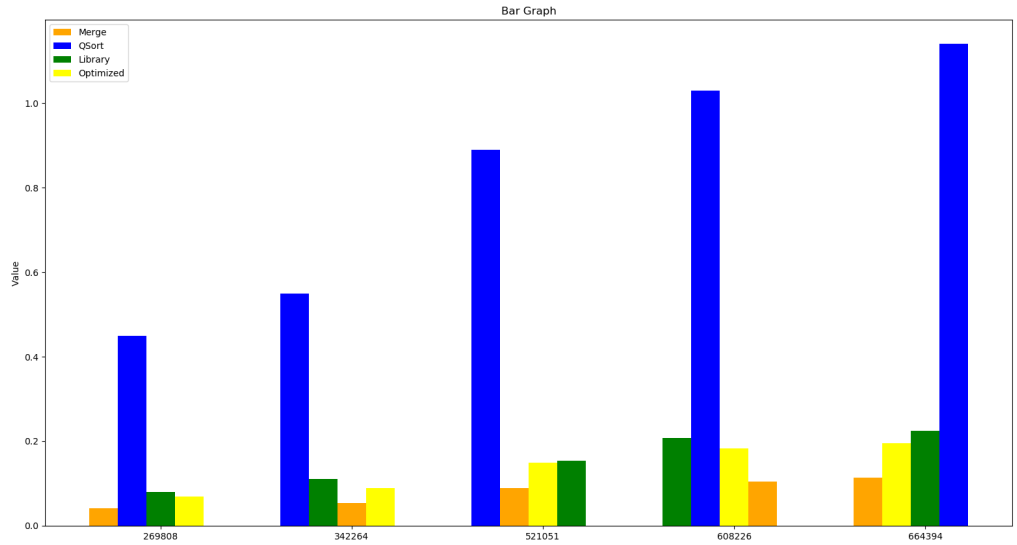


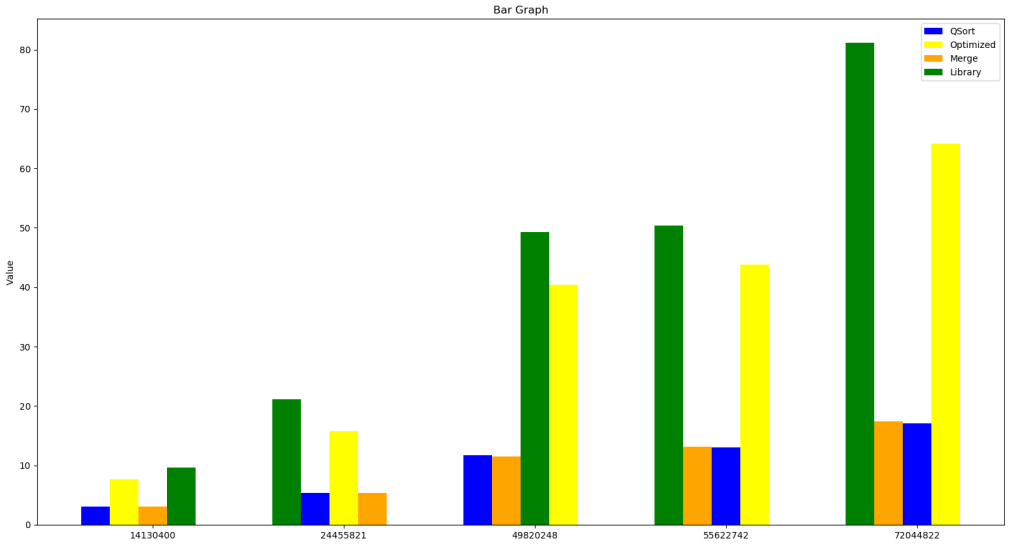
Graphs taken a few at a time for better visibility of bars for comparison:

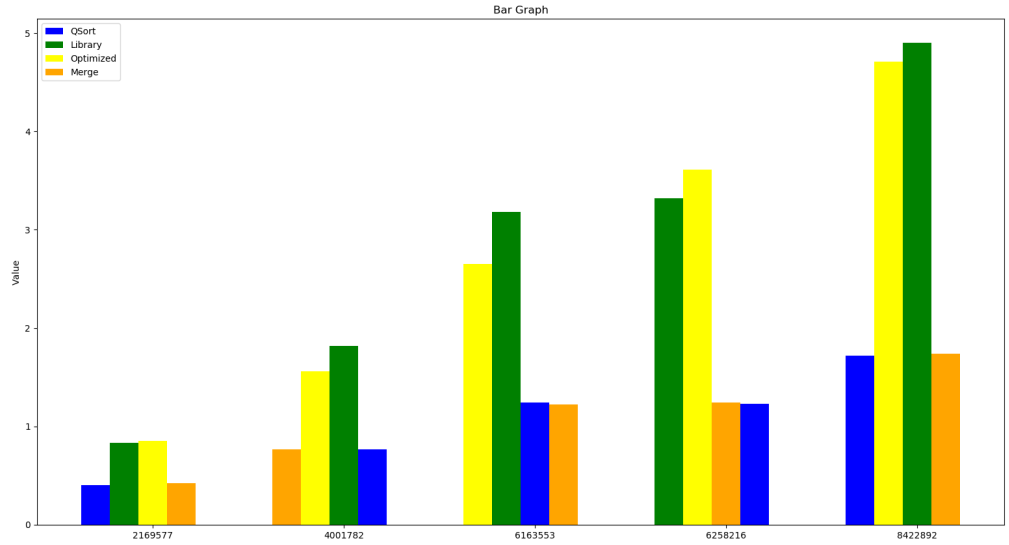




At this point, we removed insertion sort due to the huge run time (usually for arrays of size ~10^6 onwards, library sort started taking too long a time to run and test for).





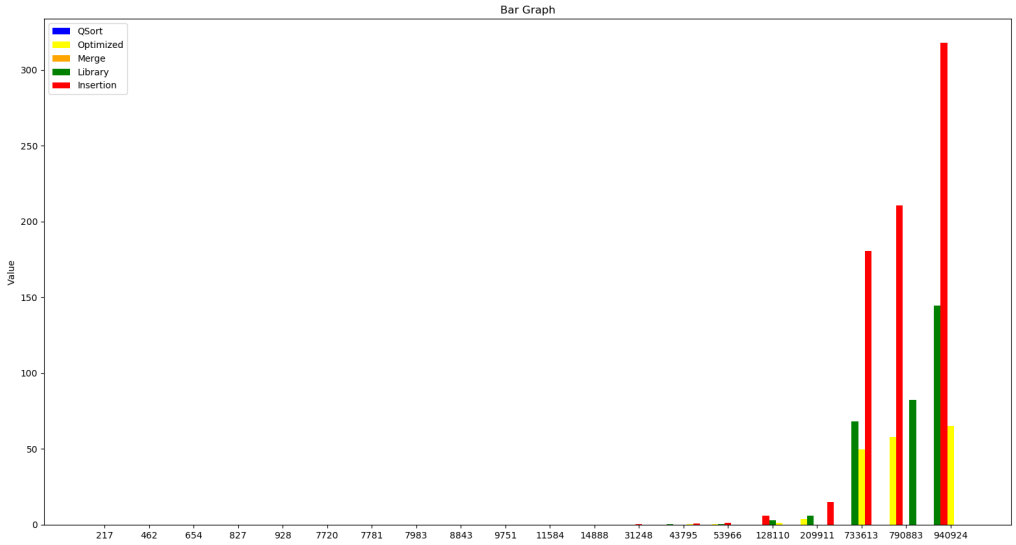


Observations:

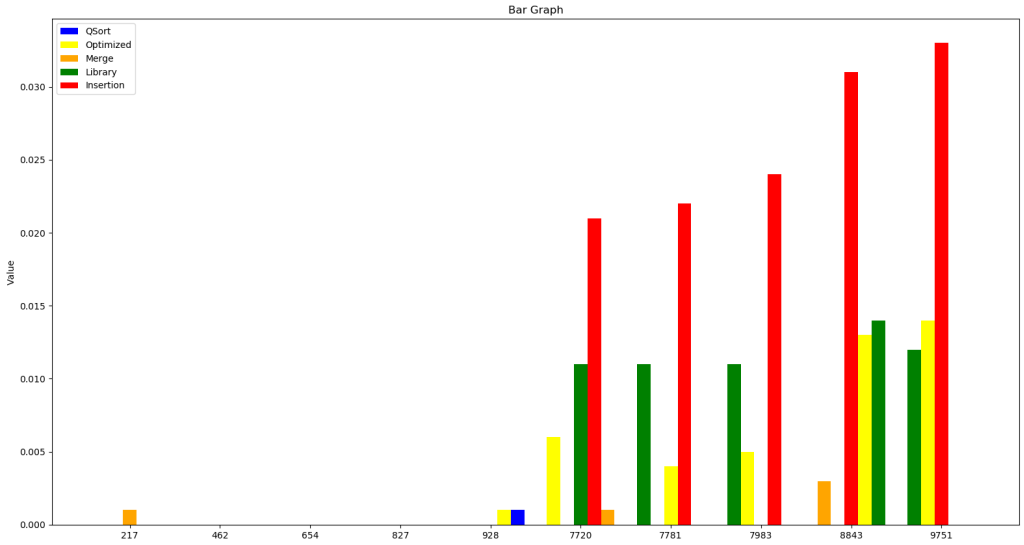
- For partially sorted arrays, the space optimisation works better i.e. Optimized Library Sort is faster than Library Sort.
- QSort is relatively much slower than the other sorting algorithms for these types of arrays.

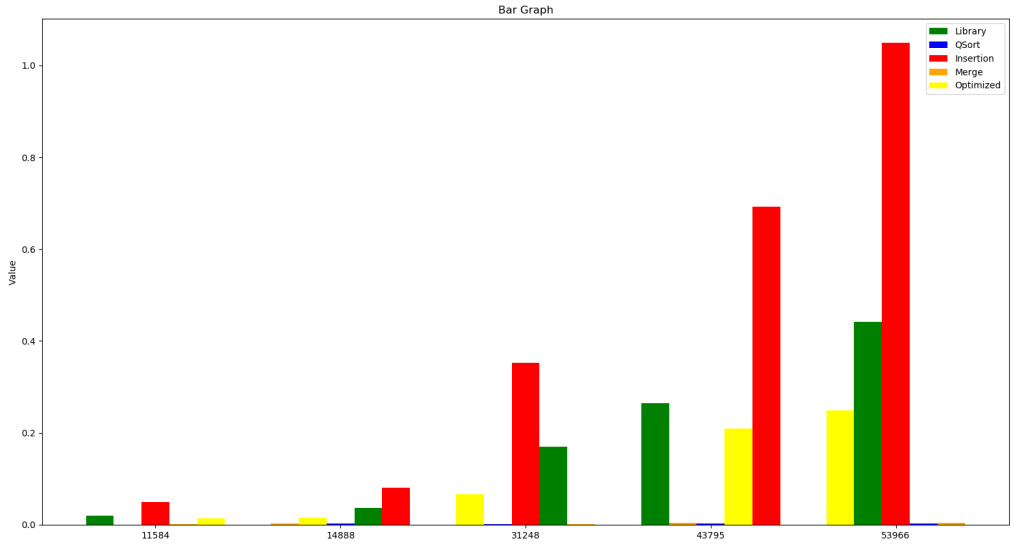
3) Reverse sorted:

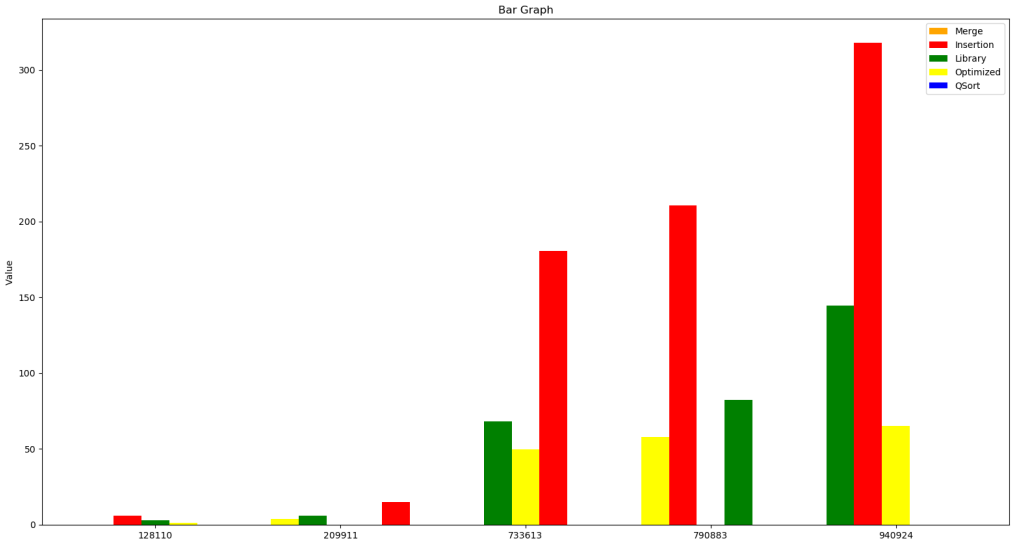
Overall graph for all sizes



Graphs taken a few at a time for better visibility of bars for comparison:







Observations:

- Optimized Library Sort performs better for all array sizes.

Conclusions based on run times:

As we can see, Library sort performs much better in almost all cases (for a significant array size where insertion sort is taking a significant amount of time) as compared to Insertion Sort, and

our Optimization reduces run times, sometimes even by a few seconds and sometimes even tremendously (especially in reverse sorted) as the size of the array increases (the magnitude of improvement depends on the type of array, but it does improve for all types as can be seen).

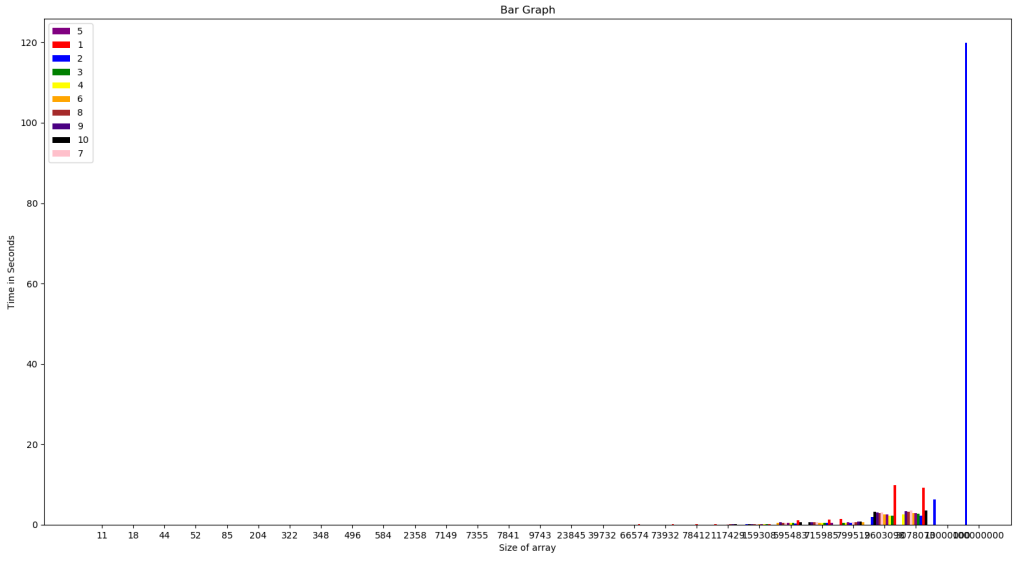
Based on Epsilon (gap size):

We have skipped the initial few graphs while going into the detailed graphs since those do not have much information as at max only 1 or 2 epsilon values are visible, while the rest have run times of almost 0. (Again, please note the scaling

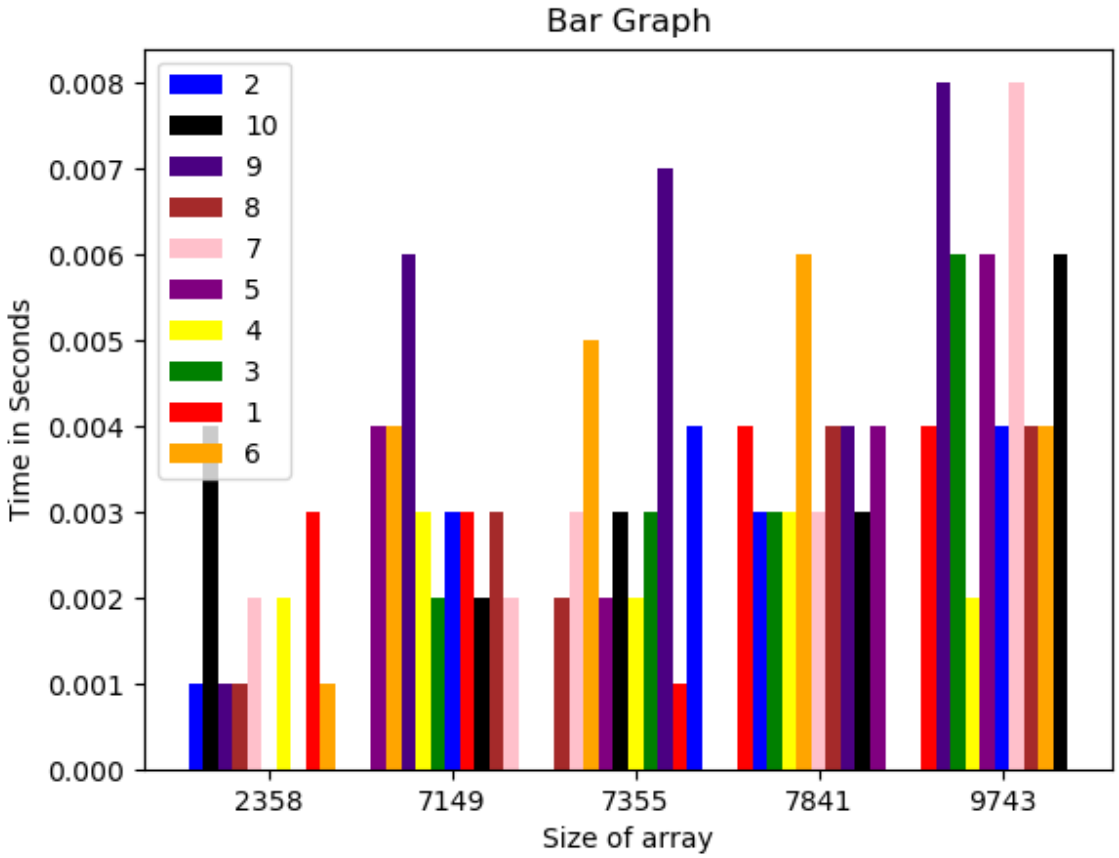
1) Fully Randomized:

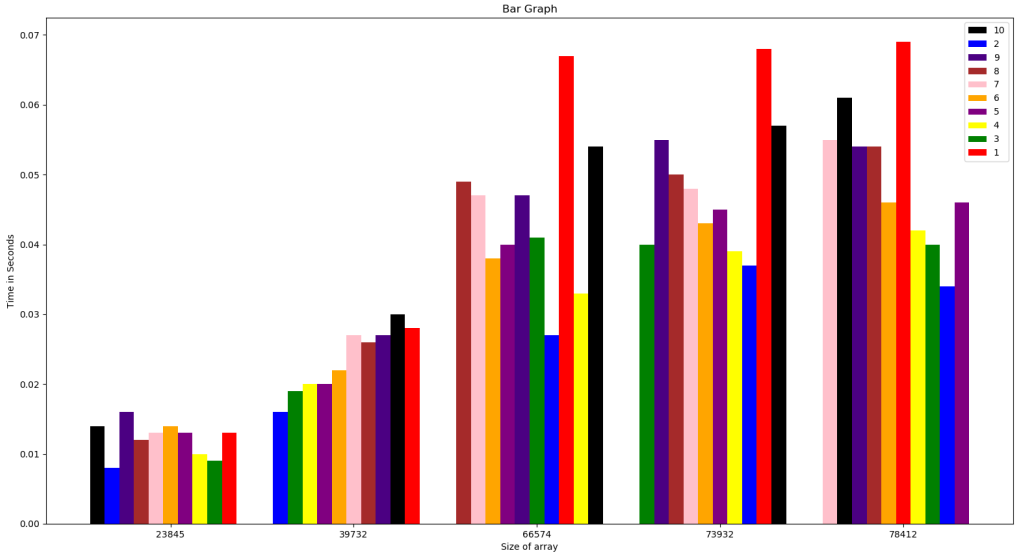
Library Sort's Epsilons:

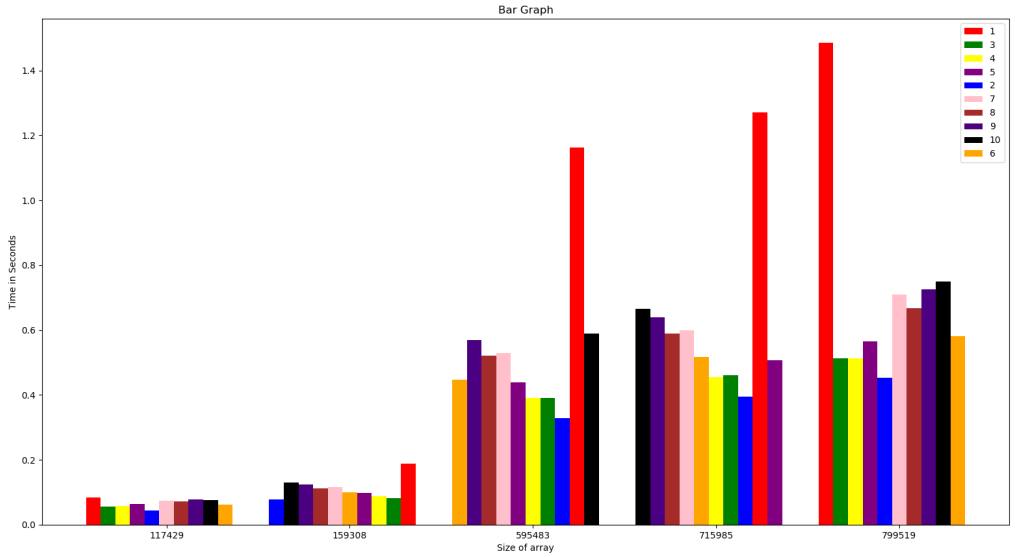
Overall graph for all sizes

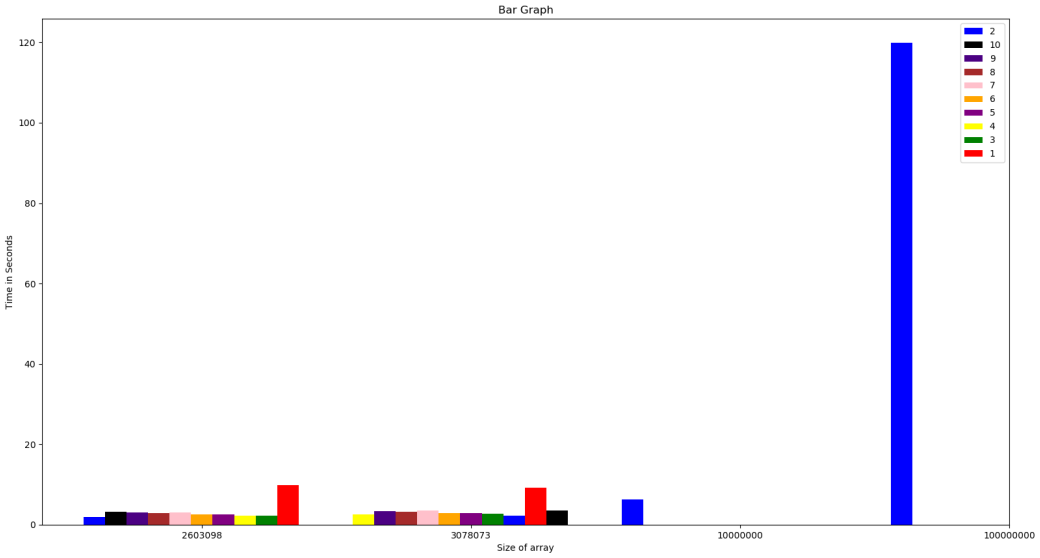


Graphs taken a few at a time for better visibility of bars for comparison:


Bar Graph
Time in Seconds
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
2
10
9
8
7
5
4
3
1
6
2358
7149
7355
7841
9743
Size of array

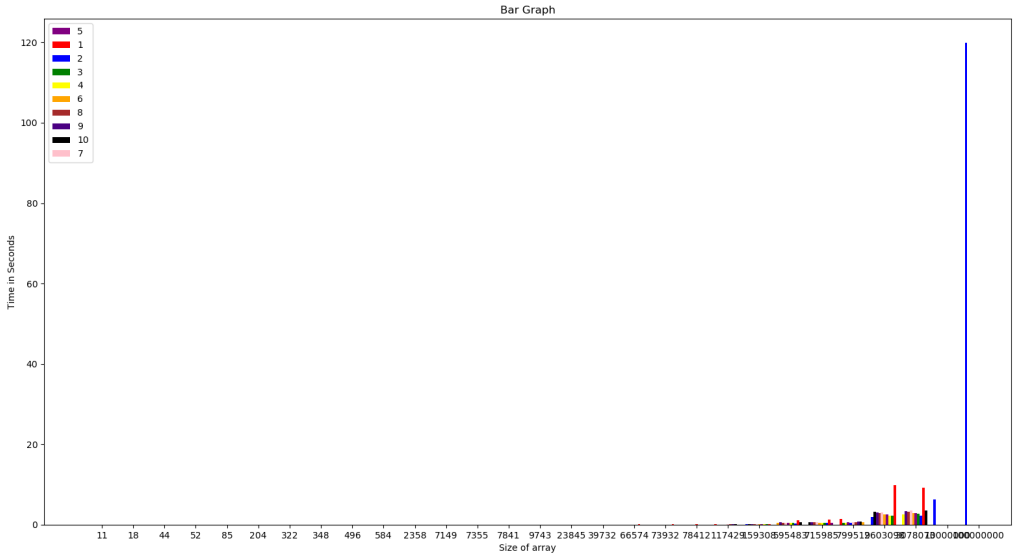

Bar Graph
Time in Seconds
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
10
2
9
8
7
6
5
4
3
1
23845
39732
66574
73932
78412
Size of array



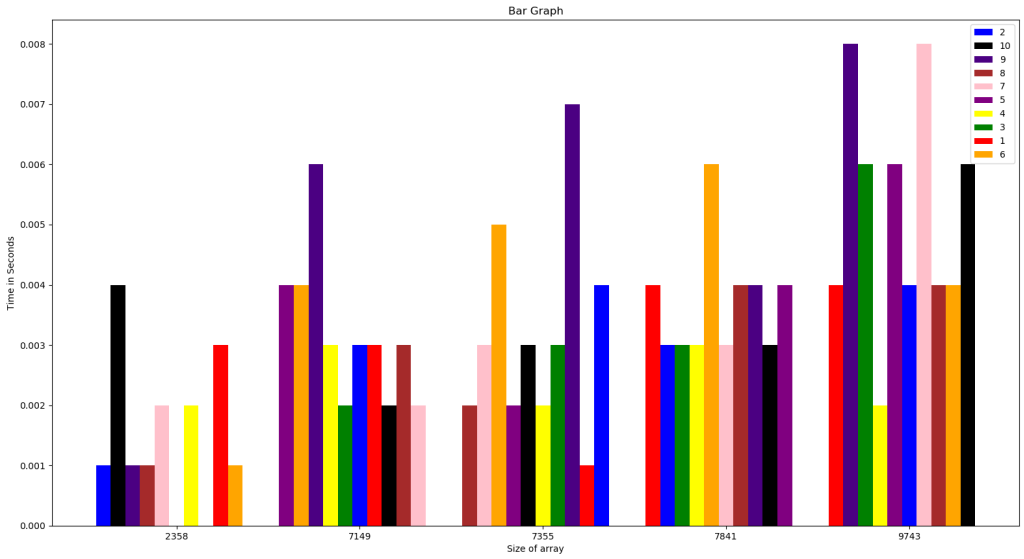


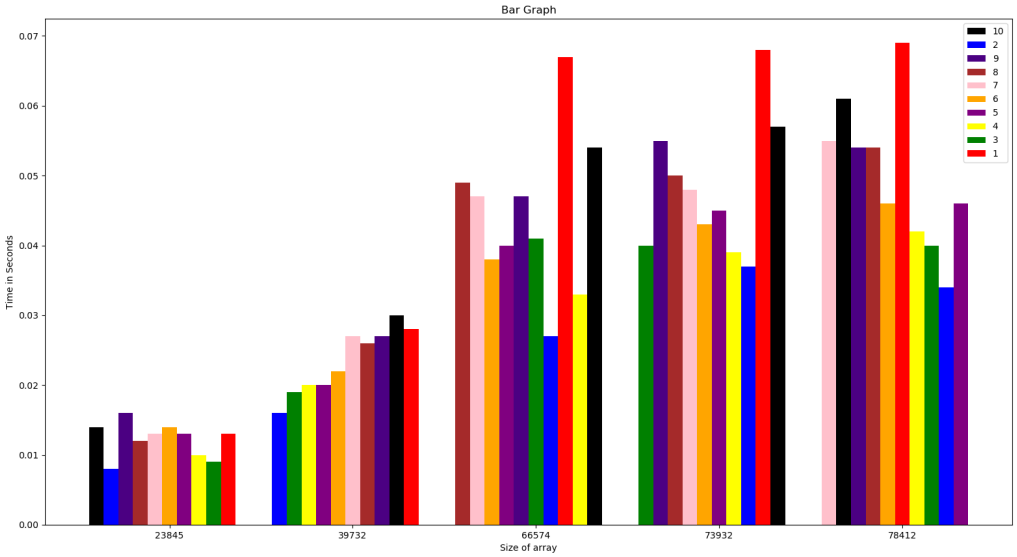
<u>Optimized Library Sort's Epsilons:</u>

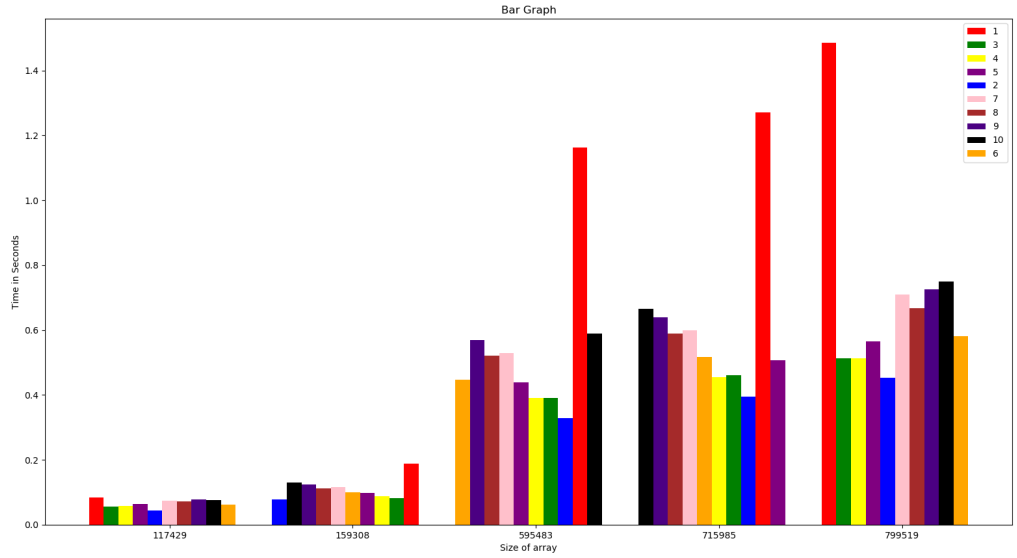
Overall graph for all sizes

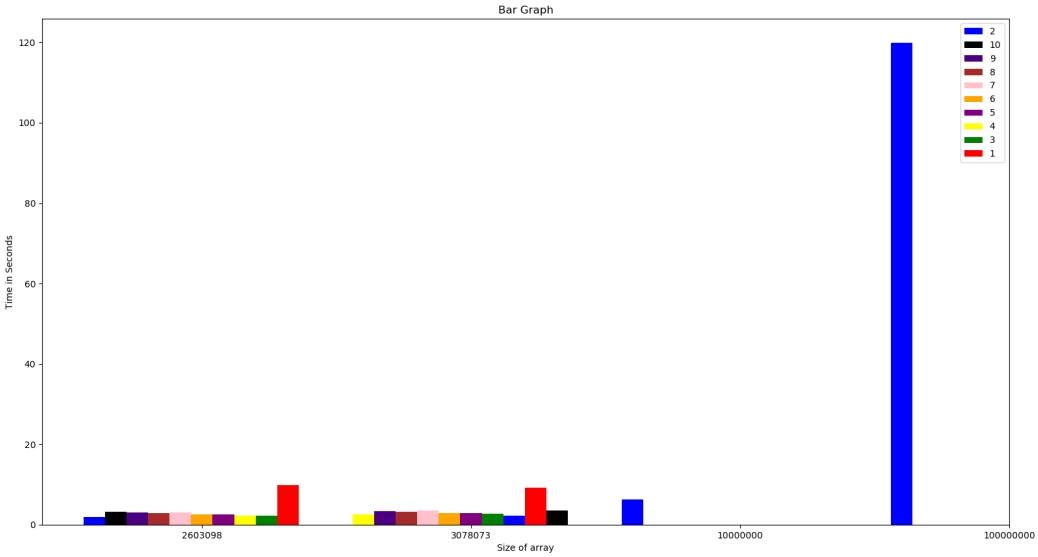


Graphs taken a few at a time for better visibility of bars for comparison:




Bar Graph
Time in Seconds
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
23845
39732
66574
73932
78412
Size of array
10
2
9
8
7
6
5
4
3
1


Bar Graph
Time in Seconds
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
117429
159308
595483
715985
799519
Size of array
1
3
4
5
2
7
8
9
10
6



Observations:

- The most optimal epsilon value depends on the quality of the array and its sortedness. (Usually between 2-10)
- Using epsilon value >=2 usually gives better time complexity than if epsilon value is 1.

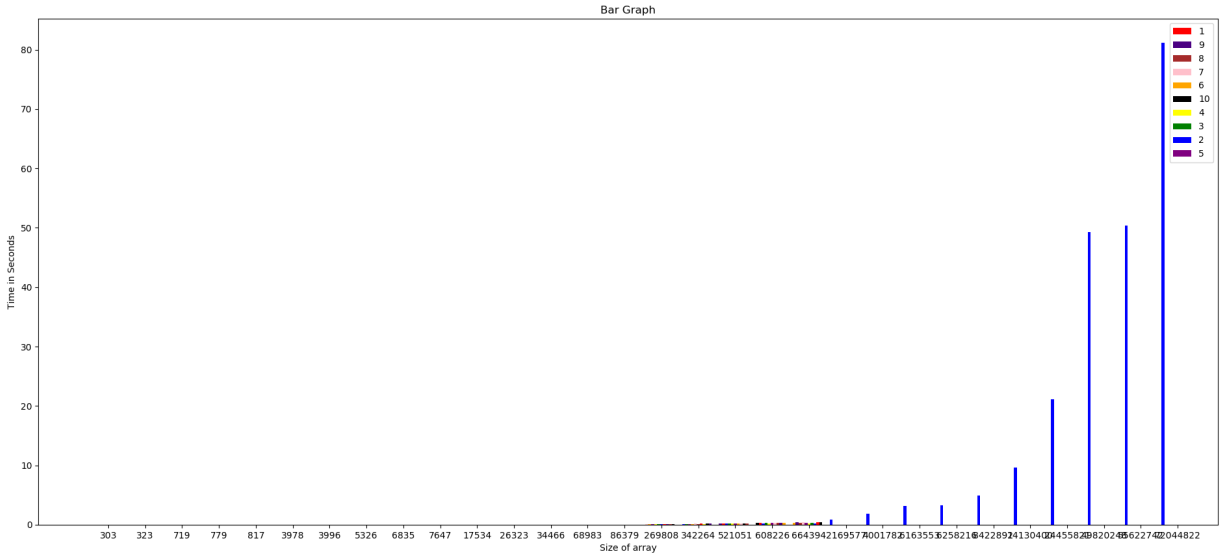
Conclusions:

As we can see in both cases, for the smaller arrays, different epsilon values get relatively better/worse in some arbitrary order, but as size keeps increasing, $\varepsilon = 1$ becomes worse and $\varepsilon = 2$ becomes better. Hence, due to the arbitrarily large run times of the last 2 arrays, we used $\varepsilon = 2$ as a predicted best value to compare with other sorts.

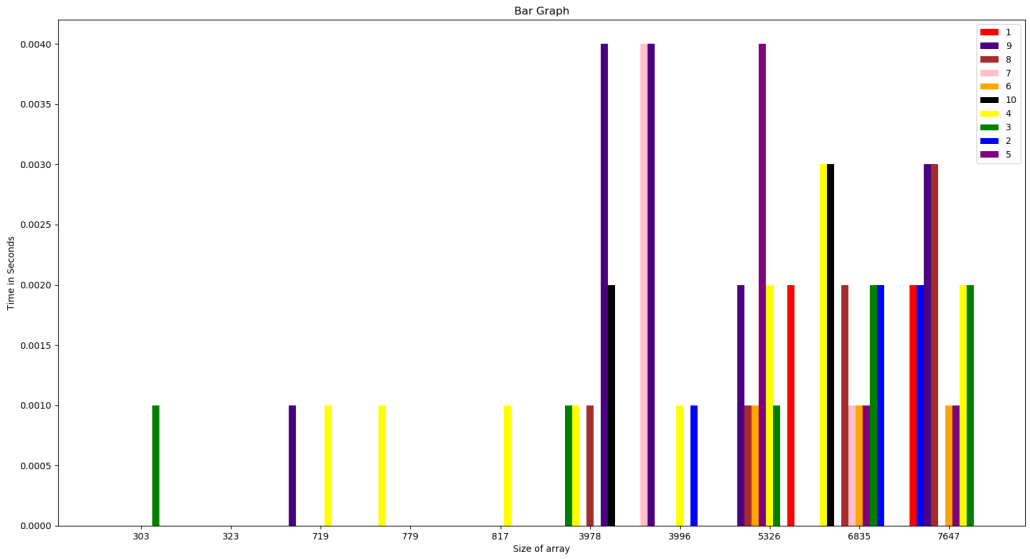
2) Partially Sorted:

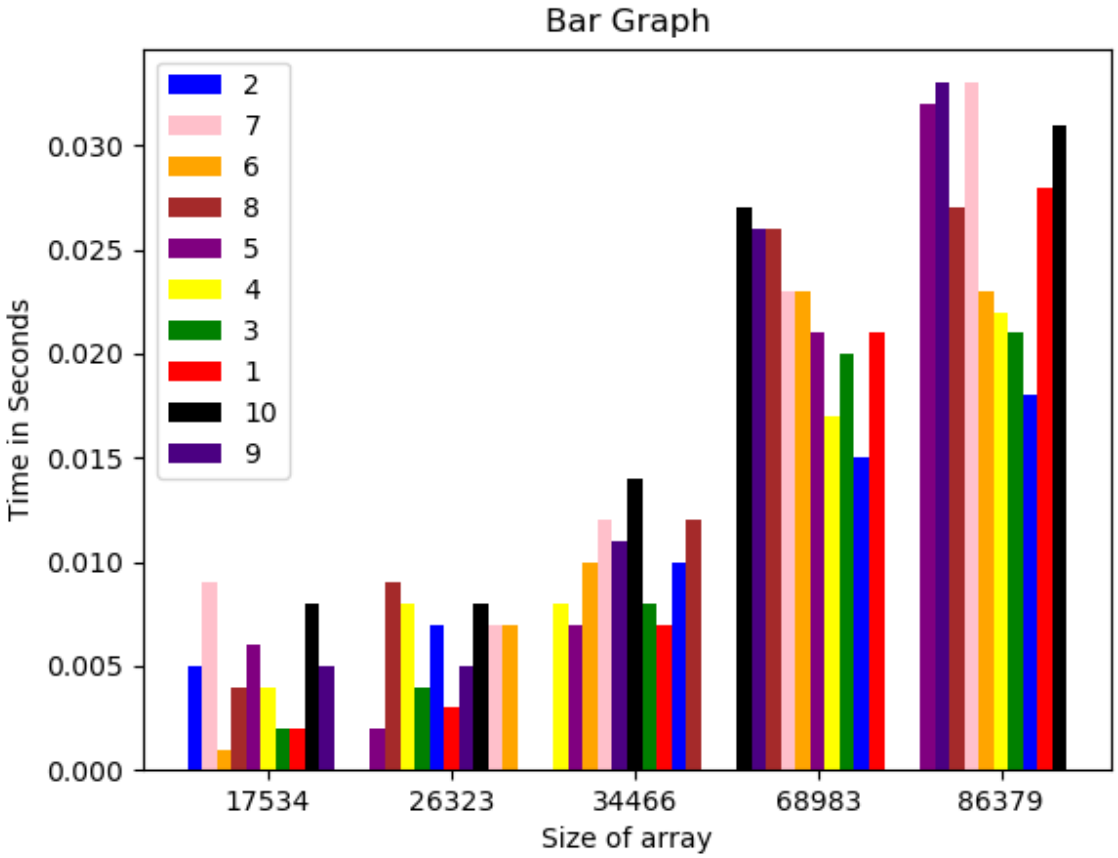
Library Sort's Epsilons:

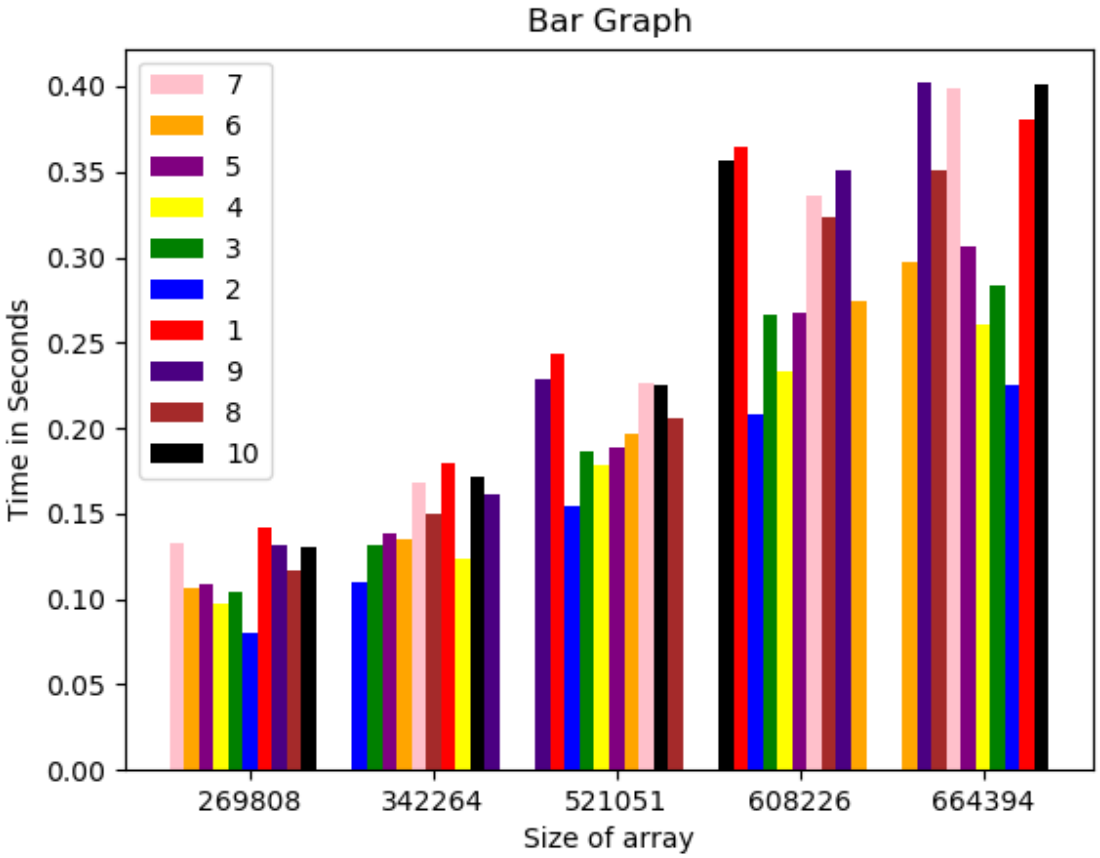
Overall graph for all sizes



Graphs taken a few at a time for better visibility of bars for comparison:



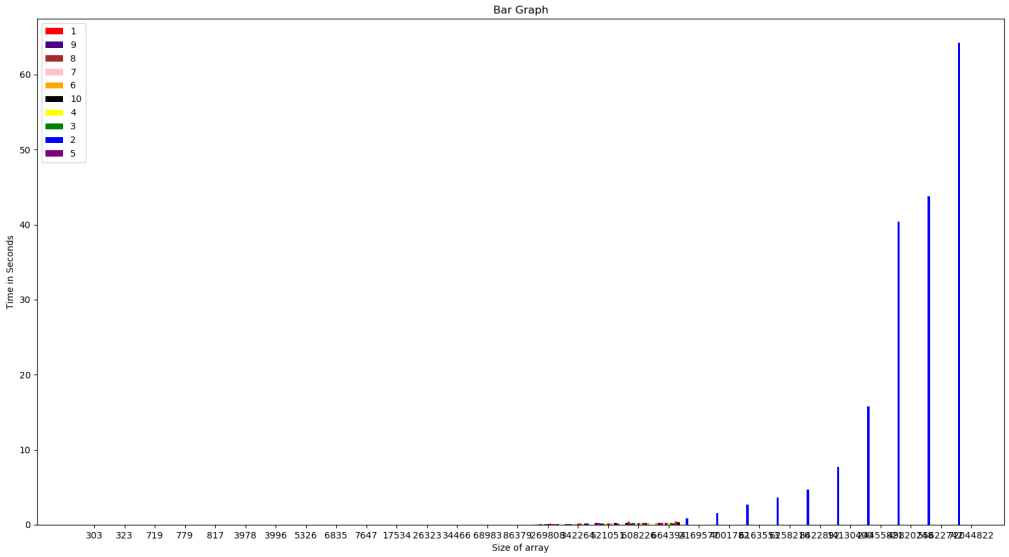

Bar Graph
2
7
6
8
5
4
3
1
10
9
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Time in Seconds
17534
26323
34466
68983
86379
Size of array



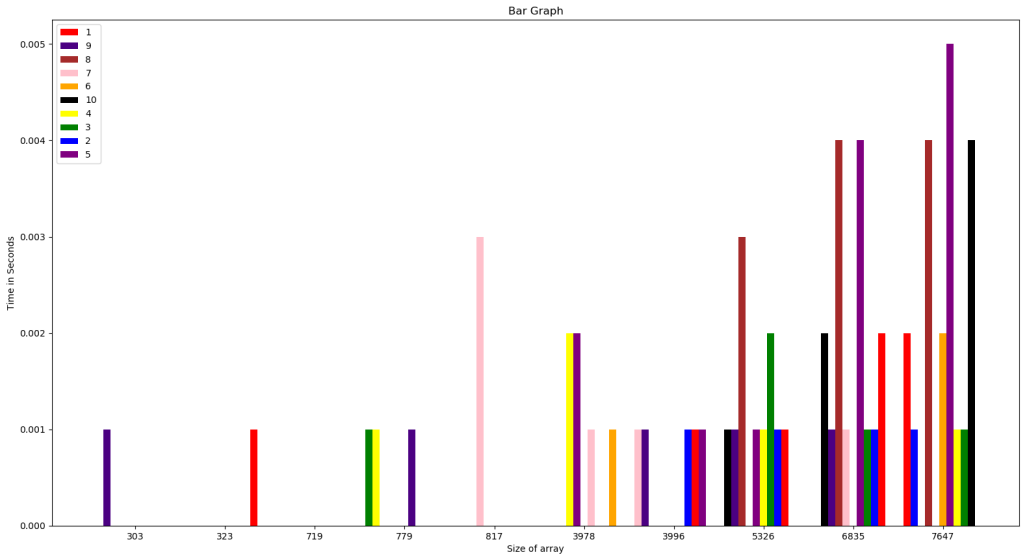
Beyond this point, due to observing a similar trend to the previous part, we only tested for $\varepsilon = 2$ due to high run times.

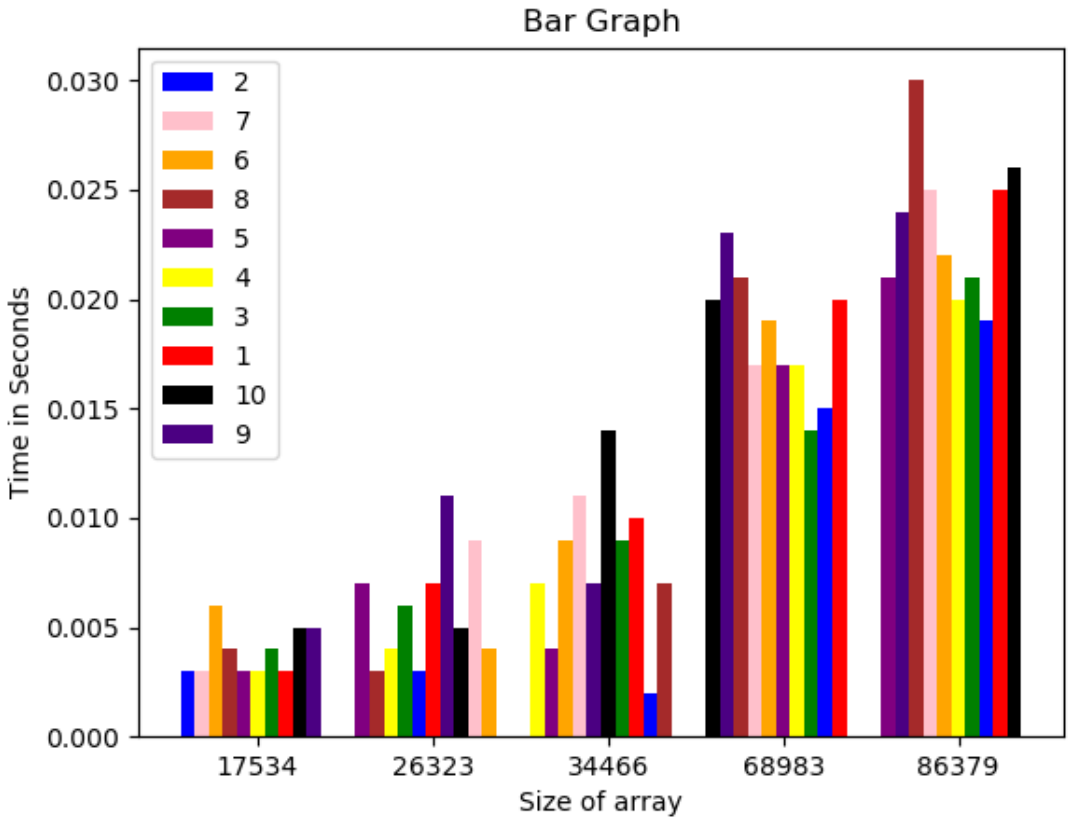
Optimized Library Sort's Epsilons:

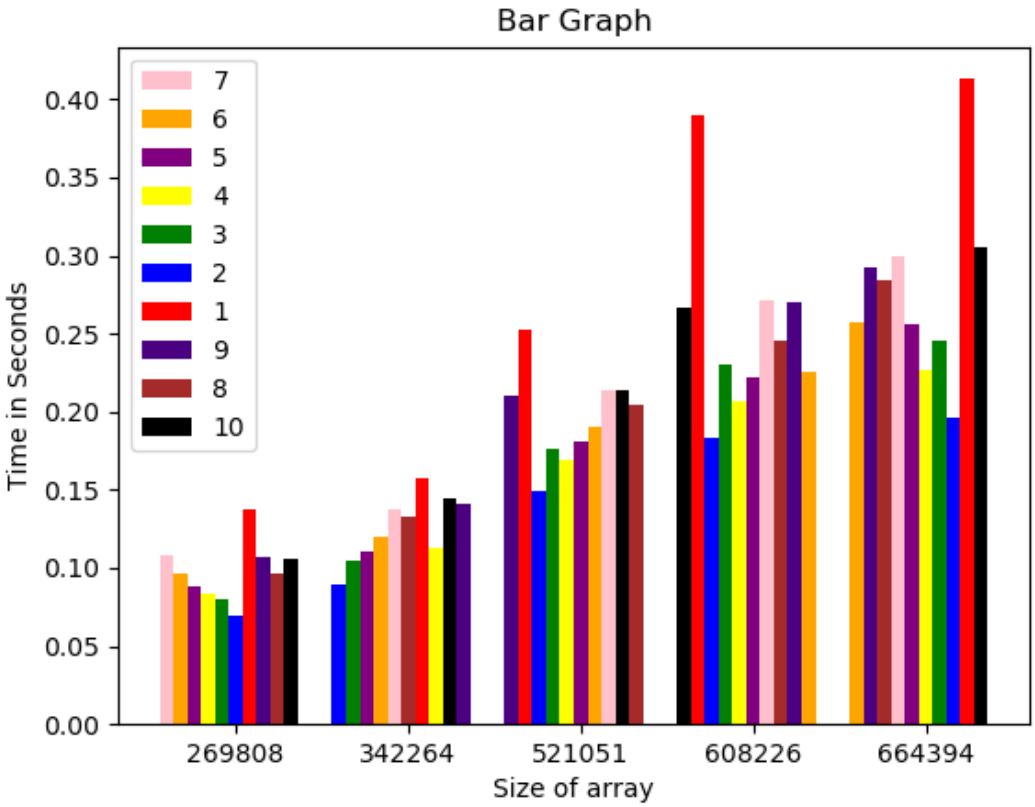
Overall graph for all sizes



Graphs taken a few at a time for better visibility of bars for comparison:




Bar Graph
2
7
6
8
5
4
3
1
10
9
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Time in Seconds
17534
26323
34466
68983
86379
Size of array



Observations:

- The most optimal epsilon value depends on the quality of the array and its sortedness. (Usually between 2-10)
- However unlike totally random arrays, epsilon 1 may have better time complexity than others in a few cases of partial sort.

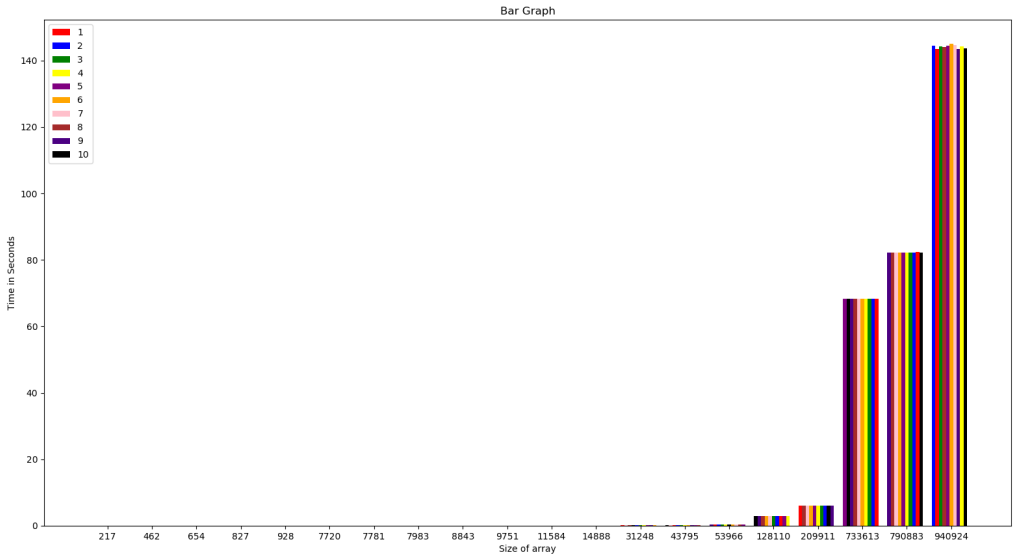
Conclusion:

We can observe a similar pattern to the fully randomized arrays, where for the smaller arrays, different epsilon values get relatively better/worse in some arbitrary order, but as size keeps increasing, $\varepsilon = 1$ becomes worse and $\varepsilon = 2$ becomes better. Hence, due to the arbitrarily large run times of the last 2 arrays, we used $\varepsilon = 2$ as a predicted best value to compare with other sorts.

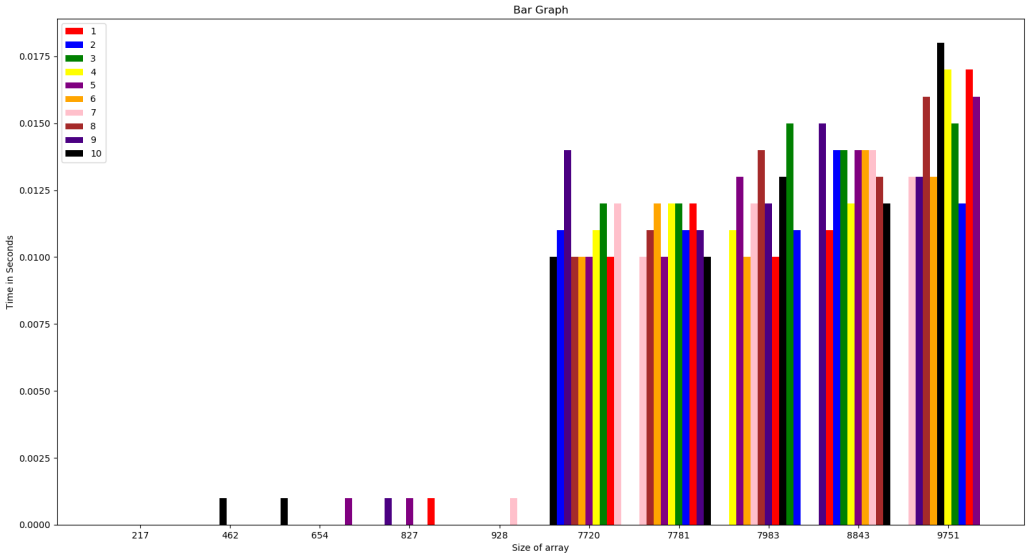
3) Reverse Sorted:

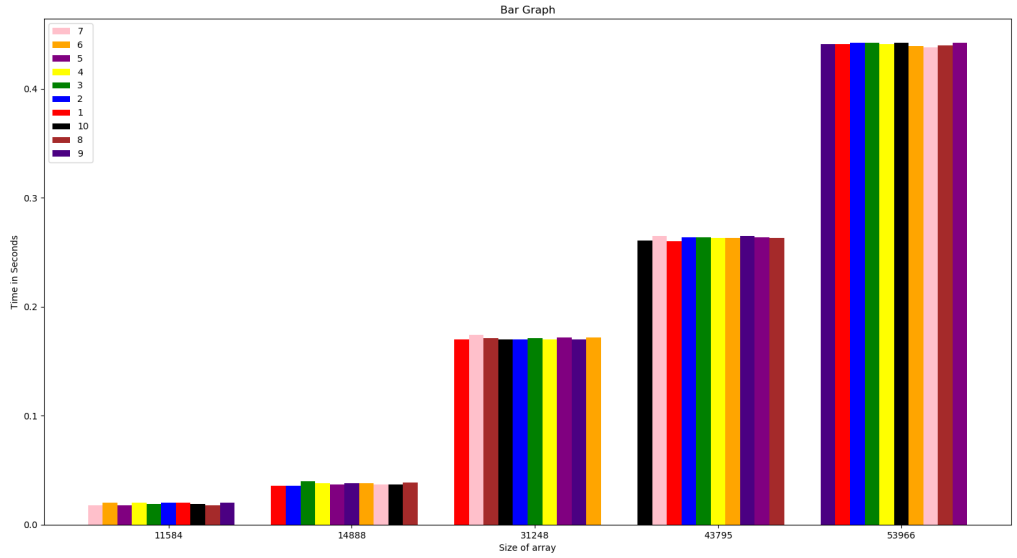
Library Sort's Epsilons:

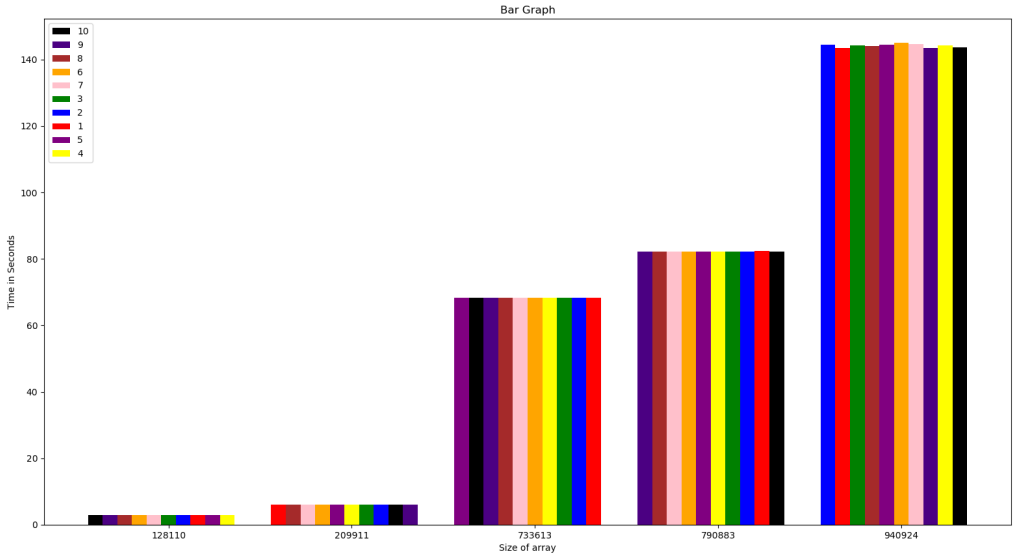
Overall graph for all sizes



Graphs taken a few at a time for better visibility of bars for comparison:





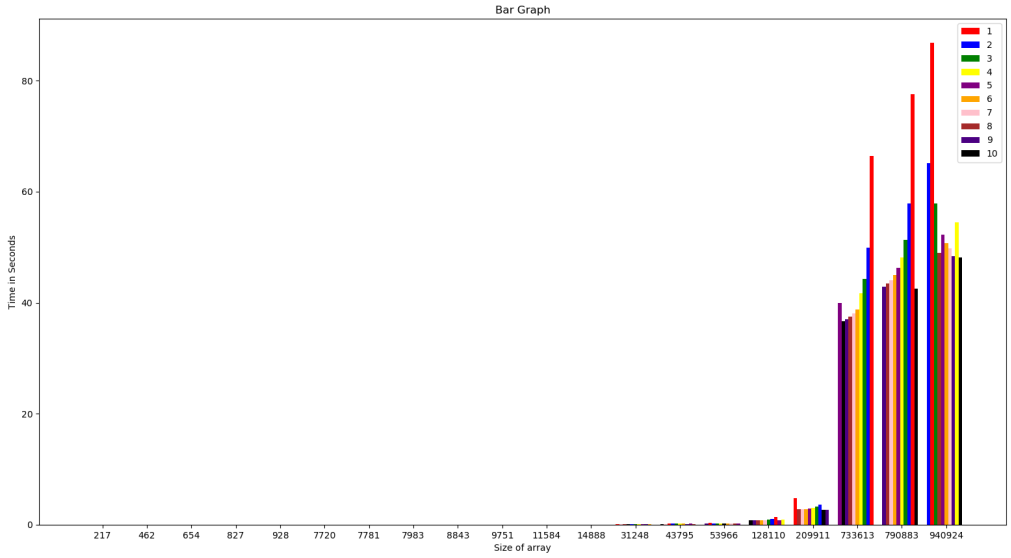


Conclusion:

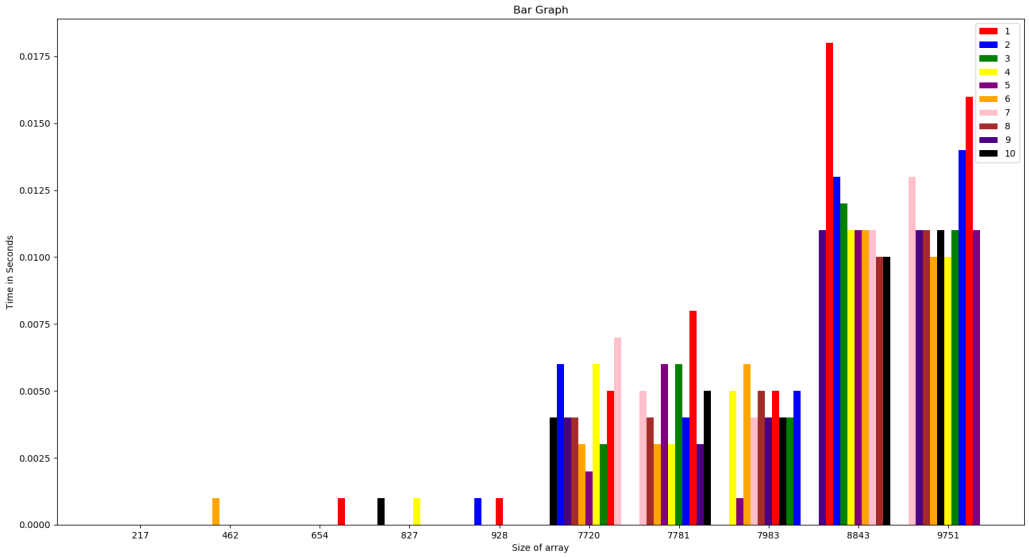
As we can see, initially, as it was before, the min/max times oscillate between different epsilons. However, as we go to bigger sizes, they get more relatively equal run times. Hence, we predict that more variation of epsilon testing is required, as higher epsilon values have slightly lower run times so if we reach a high enough value, run times should decrease.

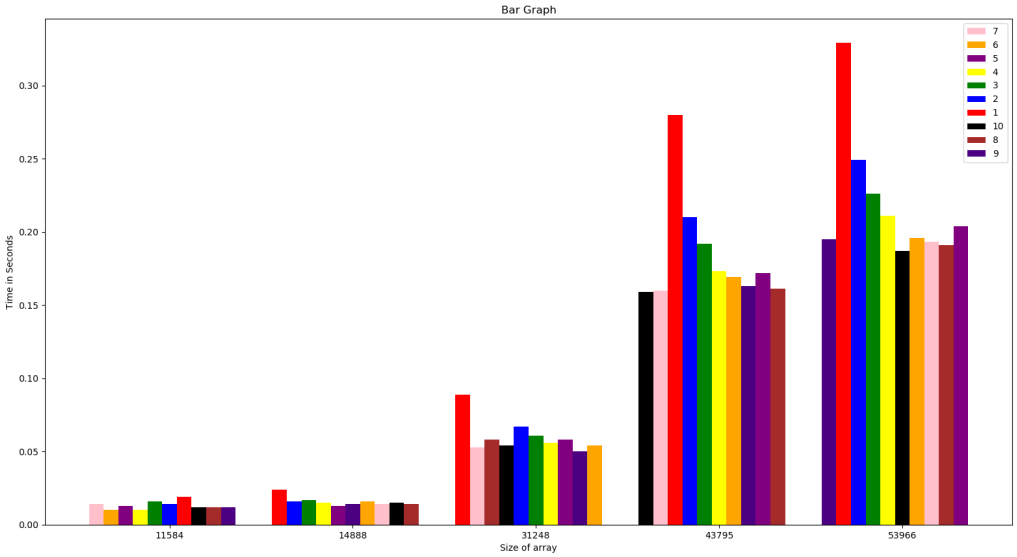
Optimized Library Sort's Epsilons:

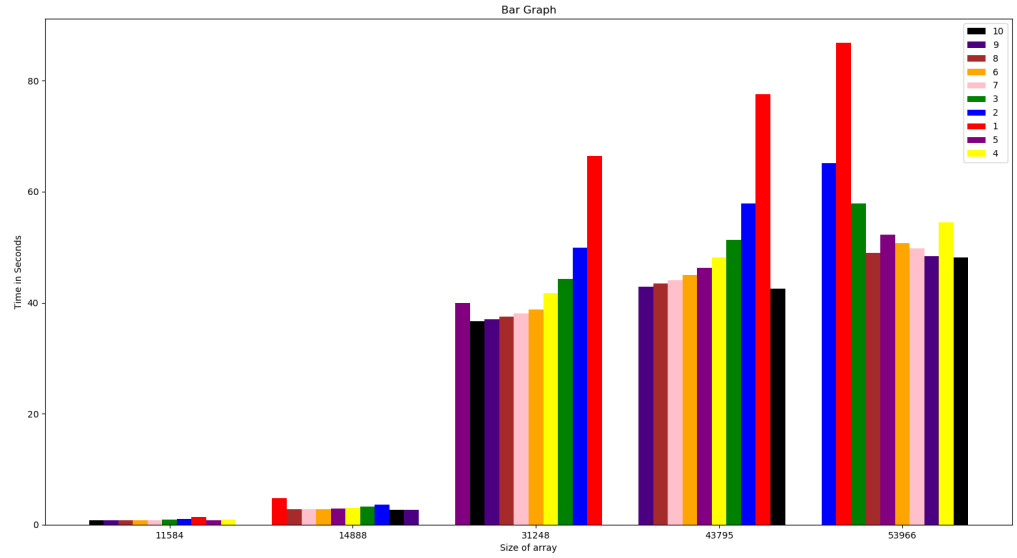
Overall graph for all sizes



Graphs taken a few at a time for better visibility of bars for comparison:







Observations:

- Since the array is reverse sorted, the computational task is almost constant for every randomisation of the array, thus for epsilon the times are similar.
- However, epsilon=1 generally takes more time for execution here too.

Conclusion:

As we can see, in the first few sizes, most epsilons have negligible run time. As we move forward, mid range values of epsilon have better run times. In the end, however, we notice the higher ranges of values have better run times, while the lower epsilons have poor run times.