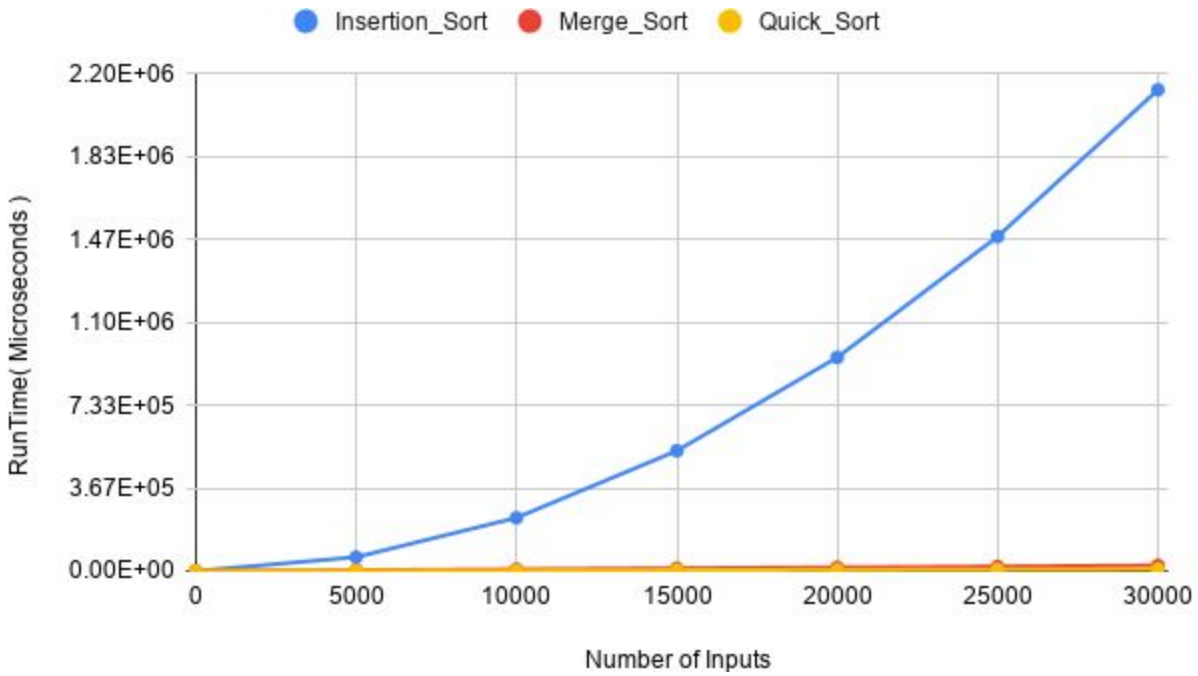
Submitted By - Sanyam Rajpal
CSCI-B-505 Applied Algorithms
(FA19 - BL - 11503)
Programming Assignment - II
WRITE-UP

I used CPP language for my code because it has faster runtime and one has to implement the algorithm from scratch helping you learn the basics thoroughly. I printed out my code in the terminal and saved them in the Google Sheets in the form of a table which I also used for plotting the graphs. I made some typical syntax errors, but not a runtime error. The observations I made have been explained below.

Plot I



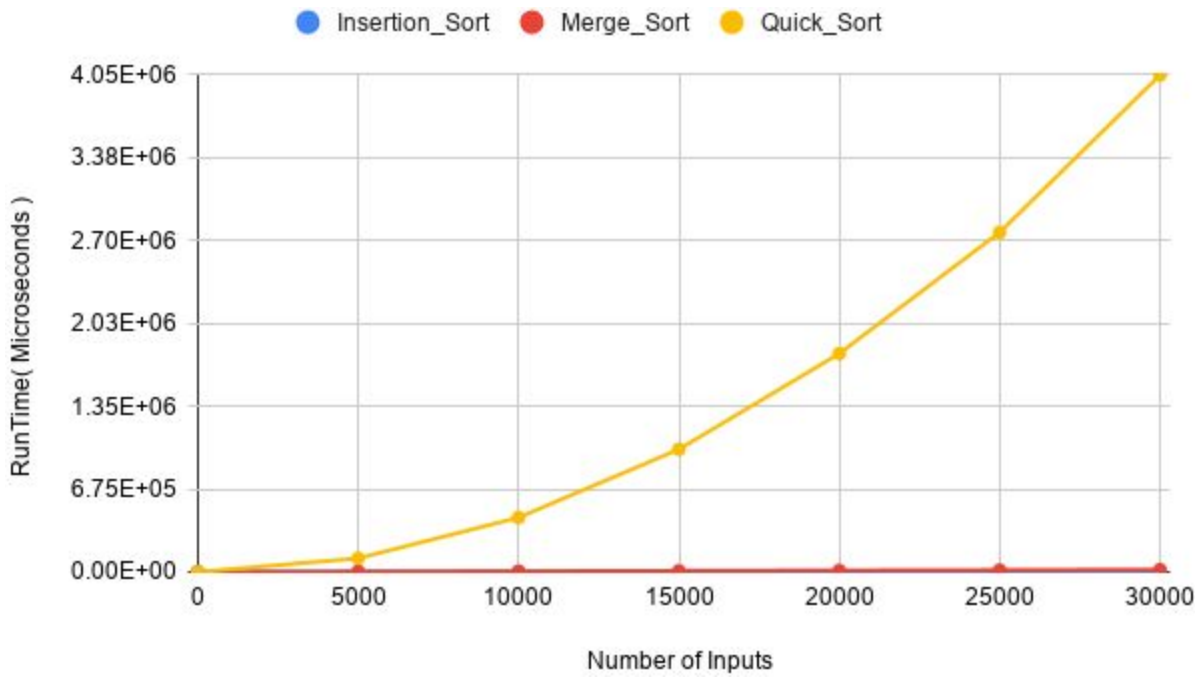
Table

Number of Inputs	Insertion_Sort	Merge_Sort	Quick_Sort
0	0.00E+00	0.00E+00	0.00E+00
5000	6.12E+04	3.71E+03	1.23E+03
10000	2.356E+05	7.72E+03	2.68E+03
15000	5.32E+05	1.18E+04	4.15E+03
20000	9.45E+05	1.59E+04	5.528E+03
25000	1.48E+06	2.01E+04	7.202E+03
30000	2.13E+06	2.45E+04	8.809E+03

This is the plot and table of an average of 3 runs for Insertion, Merge and Deterministic Quicksort with a random input array. Thus justifying the fact that Insertion Sort is $O(n^2)$ and Merge and QuickSort is O(nlogn). And trying to plot an $O(n^2)$ and O(nlog n) curve on a single graph compresses the O(nlog n) to a line parallel to y-axis because a change in that is insignificant compared to that O(n 2) and it shows that decrement of time complexity from n to log n is very significant. As the data order increases, the difference increases immensely. On observing very closely in the graph and seeing the table, quicksort is

even faster than merge sort. And the general constant factors also state that quicksort is usually faster than merge sort. Thus QuickSort > Merge Sort > Insertion Sort is the order of how fast each algorithm is.

Plot II



Table

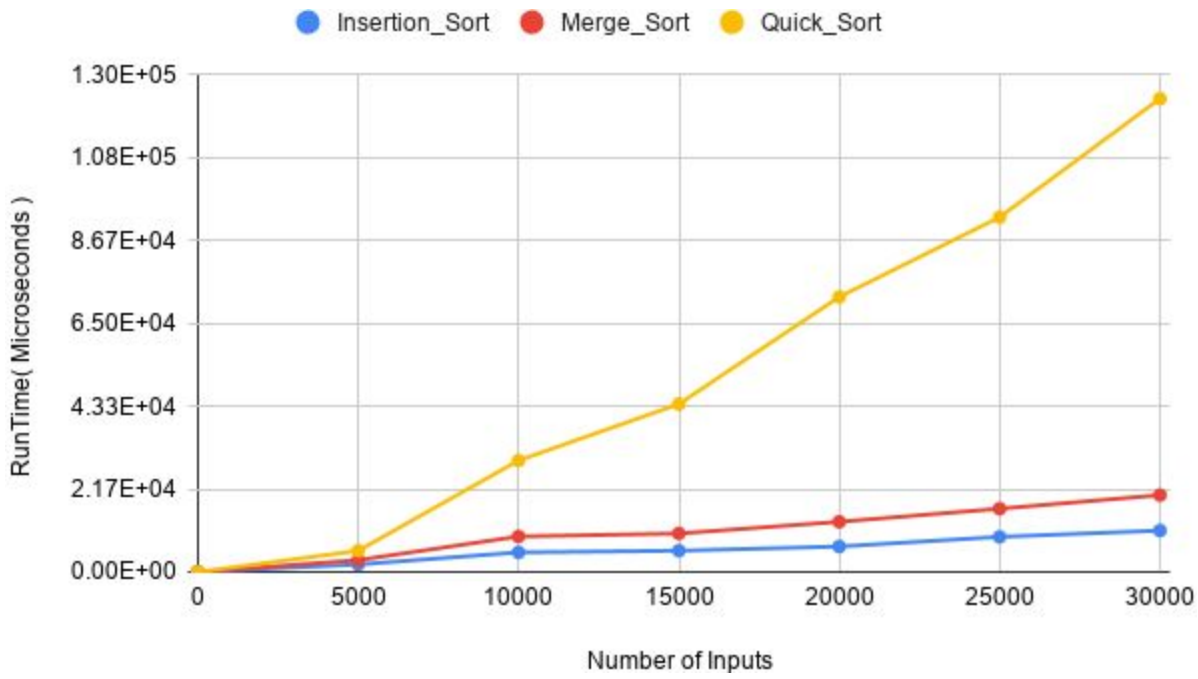
Number of Inputs	Insertion_Sort	Merge_Sort	Quick_Sort
0	0.00E+00	0.00E+00	0.00E+00
5000	1.00E+02	3.77E+03	1.11E+05
10000	1.240E+02	6.17E+03	4.43E+05
15000	1.85E+02	9.55E+03	1.00E+06
20000	2.45E+02	1.27E+04	1.78E+06
25000	3.04E+02	1.59E+04	2.77E+06
30000	3.68E+02	1.93E+04	4.05E+06

This is the plot and table for Insertion, Merge and Deterministic Quicksort with a non-decreasing array as input. As can be inferred from the plot, insertion sort performs exceptionally well than Merge and QuickSort when the input is already sorted(non-decreasing) for all orders of data. Insertion sort's time complexity is O(n) for a sorted array, but for QuickSort, it is still O(nlog n). And trying to plot an O(n) and O(nlog n) curve on a single graph compresses the O(nlog n) to a line almost parallel to the y-axis and shows log n is significant. Mergesort is way faster than quicksort. In fact, its complexity is quite close to insertion sort for sorted arrays, although not exactly O(n). Using deterministic quicksort as taught in the class made the quicksort perform slower because the pivot is fixed(last element) and the sorted array isn't the best case for quicksort. Thus Insertion Sort > Merge Sort > QuickSort is the order of how fast each algorithm is.

Number of Inputs	Insertion_Sort	Merge_Sort	Quick_Sort
0	0.00E+00	0.00E+00	0.00E+00
5000	1.18E+05	3.08E+03	8.64E+04
10000	5.193E+05	6.31E+03	3.48E+05
15000	1.06E+06	9.66E+03	7.76E+05
20000	1.89E+06	1.31E+04	1.38E+06
25000	2.96E+06	1.64E+04	2.15E+06
30000	4.25E+06	1.99E+04	3.09E+06

This is the plot and table for Insertion, Merge and Deterministic Quicksort with a non-increasing array as input. As can be inferred from the plot, merge sort performs way better than insertion sort and quicksort. Quicksort performs slightly better than insertion sort which can be inferred from the table and the graph. Merge Sort performs exceptionally good for both the ascending and descending array. Insertion sort, on the other hand is fastest for sorted array and slowest for a descendingly sorted array. Quick Sort doesn't perform well on non-increasing arrays either and its time complexity approaches $O(n^2)$. And since time complexity of merge sort is O(nlogn) it compresses to a straight line when plotted along with $O(n^2)$. Thus Merge Sort > QuickSort > Insertion Sort is the order of how fast each algorithm is.

Plot IV



Table

Number of Inputs	Insertion_Sort	Merge_Sort	Quick_Sort
0	0.00E+00	0.00E+00	0.00E+00
5000	1.98E+03	3.11E+03	5.55E+03
10000	5.137E+03	9.30E+03	2.92E+04
15000	5.57E+03	1.01E+04	4.39E+04
20000	6.67E+03	1.31E+04	7.19E+04
25000	9.19E+03	1.66E+04	9.28E+04
30000	1.08E+04	2.01E+04	1.24E+05

This is the plot and table of an average of 3 runs for Insertion, Merge and Deterministic Quicksort with a non-decreasing array as input with 50 swaps between random indexes. As can be inferred from the plot and the table, insertion sort is performing even faster than merge sort, just like in Plot II. Insertion sort performs close to the O(n) when the array is almost sorted. Quicksort on the other hands performs the slowest just like in Plot II because its complexity is close to O(nlogn). Merge sort performs close to Insertion sort, and as you can see they are both performing almost similar, unlike in the case of Plot II Table where insertion sort is at least 1 order faster than merge sort. So, those 50 swaps in the sorted array made the time difference even smaller. We are comparing it to Plot II because it's a derivative version of that with just 50 swaps where the order of data is 10^4. Thus Insertion Sort > Merge Sort > QuickSort is the order of how fast each algorithm is.

Input-5

For Input-5. We had 100,000 input values in the range [1 , 50]. Since 50 << 100,000 there will be plenty of repeated values. The runtime for Insertion sort is **2.50283e+07** microseconds, runtime for Merge sort is **8.0076e+04** microseconds and the runtime for Deterministic QuickSort is **3.95176e+05** microseconds. Insertion sort is at least slower by order 2 of time than Quick and Merge Sort. Merge Sort is almost 1 order of time faster than QuickSort. So, when there are plenty of repeated values merge sort outperforms quick sort. So, we can infer that when almost all elements of an array are frequently occurring in an array merging turns out to be better than partitioning using pivot.

Conclusion

Insertion Sort is $O(n^2)$ algorithm in general whose best case is when an array is fully sorted and thus making its time complexity O(n). So, I would go for this algorithm when the fact that the array is almost sorted in known because its space complexity is also O(1).

Merge sort algorithm although is O(n2) but its space complexity is O(n). It can be useful when space is not an issue and array is an either non-increasing or non-decreasing array. Because it outperforms quick sort for such cases. And except for the non-decreasing arrays it always outperforms insertion sort.

Deterministic quicksort is a good algorithm. It performs better than both the algorithms stated above in terms of time as well as memory for a general array. But for cases when an array is non-increasing or non-decreasing the array or any of their derivatives it usually performs slower than merge and insertion sort. But it has the benefit of not using much memory. For non-increasing arrays, it performs better than insertion sort and for that, it can be chosen, because both have similar space complexity for memory.

//SOURCE CODE

```
// Since these are the common functions for all the different main functions for all plots, I wrote them
//once on the top.
//including required libraries
#include <iostream>
#include <vector>
#include <algorithm>
#include <time.h>

// to not use std again and again when needed
using namespace std ;
// function to print array
void print_array( vector< int > input ){
  for( auto i : input )
    cout << i << " " ;
  cout << endl ;
}

// insertion sort algorithm implemented as a function
void insertion_sort( vector< int > input , int size ){
```

```
  int j , key ;
  for( int i = 1 ; i < size ; ++i ){
    key = input[ i ] ;
    j = i ;
    // ( j-- ) is true for all values of j greater than 0. Plus it narrows down the loop line by 1 by doing the
    //decrement
    while( ( j-- ) && ( input[ j ] > key ) )
      input[ j + 1 ] = input[ j ] ;
    input[ j + 1 ] = key ;
  }

  // cross checking that the array has been sorted
  //cout << "Sorted Array" << endl ;
  //print_array( input ) ;
}

// merge algorithm implemented as a function
void merge( vector< int > &input , int left , int mid , int right ){
  // get size of 2 arrays to be merged.
  int size_l = mid - left + 1 ;
  int size_r = right - mid ;

  // initializing variables
  int i = 0 ;
  int j = 0 ;
  int k = left ;
  // initialize 2 arrays with size equal to size_l and size_r respectively.
  vector< int > arr_l( size_l ) ;
  vector< int > arr_r( size_r ) ;
  while( i < size_l ){
    arr_l[ i ] = input[ i + left ] ;
    ++i ;
  }
  while( j < size_r ){
    arr_r[ j ] = input[ j + 1 + mid ] ;
    ++j ;
  }
  i = 0 ;
  j = 0 ;

  while( ( i < size_l ) && ( j < size_r ) ){
    if( arr_l[ i ] <= arr_r[ j ] )
      input[ ( k++ ) ] = arr_l[ ( i++ ) ] ;
    else
      input[ ( k++ ) ] = arr_r[ ( j++ ) ] ;
  }

  while( i < size_l ){
```

```
    input[ ( k++ ) ] = arr_l[ ( i++ ) ] ;
  }

  while( j < size_r ){
    input[ ( k++ ) ] = arr_r[ ( j++ ) ] ;
  }
}

// merge sort algorithm implemented as a function
void merge_sort( vector< int > &input , int left , int right ){
  if( left < right ){
    int mid = left + ( right - left ) / 2 ;
    merge_sort( input , left , mid ) ;
    merge_sort( input , mid + 1 , right ) ;
    merge( input , left , mid , right ) ;
  }
}

// calls merge_sort function and prints array after being sorted.
// it also helps in passing &vector and not replacing values in the input
// function so that it can be used for sorting algorithms written further in
// main function.
void merge_print( vector< int > input , int left , int right ){
  merge_sort( input , left , right ) ;
  // cross checking that the array has been sorted
  //cout << "Sorted Array" << endl ;
  //print_array( input ) ;
}
// function to swap values of the array elements
// * is uses so that address is not changed and only the values at those addresses are changed which is
// what we want
// the value of array elements are changed
void swap( int *x_pointer , int *y_pointer ){
  int temp = *x_pointer ;
  *x_pointer = *y_pointer ;
  *y_pointer = temp ;
}

// partition algorithm implemented as a function
int partition( vector< int > &input , int low , int high ){

  // making the deterministic quick_sort random.
 //  int i = low + rand( ) % ( high - low + 1 ) ;
 // swap( &input[ i ] , &input[ high ] ) ;

  int pivot = input[ high ] ;
  int i = low - 1 ;
```

```
  for( int j = low ; j < high ; ++j )
    if( input[ j ] < pivot )
      swap( &input[ ++i ] , &input[ j ] ) ;

  swap( &input[ ++i ] , &input[ high ] ) ;
  return i ;
}

// deterministic quick sort algorithm implemented as a function
void quick_sort( vector< int > &input , int low , int high ){
  if( low < high ){
    int part = partition( input , low , high ) ;
    quick_sort( input , low , part - 1 ) ;
    quick_sort( input , part + 1 , high ) ;
  }
}

// calls quick_sort function and prints array after being sorted.
void quick_print( vector< int > input , int low , int high ){
  quick_sort( input , low , high ) ;
  // cross checking that the array has been sorted
  //cout << "Sorted Array" << endl ;
  //print_array( input ) ;
}

// Input-1 Plot-1 Main Function
// main function which calls the respective algorithm for sorting
int main(){
  // initializing variables and arrays
  vector< int > input ;
  int size , j ;
  // asking user for input of the number of elements
  cout << "Give the number of elements you would like the randomly generated input to have." << endl ;
  cin >> size ;
  // initializing variables with values
  double sum_ins = 0.0 ;
  double sum_merge = 0.0 ;
  double sum_quick = 0.0 ;
  // time variables
  clock_t begin , end ;
  for( int i = 0 ; i < 3 ; ++i ){
    // clearing input vector from the previous values stored so that a new set of random values can be
    // added and later sorted to make 3 random sortings distinct
    // pushing size number of random values in range [ 1 , size ]
    input.clear( ) ;
    for( j = 0 ; j < size ; ++j )
      input.push_back( rand( ) % ( size ) + 1 ) ;
      // cross checking random arrays
```

```
    //cout << "Random Array" << endl ;
    //print_array( input ) ;
    begin = clock( ) ;
    // here & operator isn't used so the sorted array won't be saved at the address we are accessing
    // from input[ ] in the main function
    insertion_sort( input , size ) ;
    end = clock( ) ; // calculating runtime of sorting and adding it to the sum of runtimes
    sum_ins += double( end - begin ) ;
    cout << "Execution Time for " << ( i + 1 ) << " time for insertion sort is " << double( end - begin ) << "
microseconds" << endl ;
    // cross checking random arrays
    //cout << "Random Array" << endl ;
    //print_array( input ) ;
    begin = clock( ) ;
    // here & operator isn't used so the sorted array won't be saved at the address we are accessing
    // from input[ ] in the main function
    merge_print( input , 0 , size - 1 ) ;
    end = clock( ) ;
    // calculating runtime of sorting and adding it to the sum of runtimes
    sum_merge += double( end - begin ) ;
    cout << "Execution Time for " << ( i + 1 ) << " time for merge sort is " << double( end - begin ) << "
microseconds" << endl ;
    // cross checking random arrays
    //cout << "Random Array" << endl ;
    //print_array( input ) ;
    begin = clock( ) ;
    // here & operator isn't used so the sorted array won't be saved at the address we are accessing
    // from input[ ] in the main function
    quick_print( input , 0 , size - 1 ) ;
    end = clock( ) ;
    // calculating runtime of sorting and adding it to the sum of runtimes
    sum_quick += double( end - begin ) ;
    cout << "Execution Time for " << ( i + 1 ) << " time for deterministic quick sort is " << double( end - begin ) <<
" microseconds" << endl ;
  }
  // printing average of the runtimes of insertion and selection sort
  cout << "Average runtime for all 3 sortings of insertion sort is " << ( sum_ins / 3 ) << " microseconds" <<
endl ;
  cout << "Average runtime for all 3 sortings of merge sort is " << ( sum_merge / 3 ) << " microseconds" <<
endl ;
  cout << "Average runtime for all 3 sortings of deterministic quick sort is " << ( sum_quick / 3 ) << "
microseconds" << endl ;
}
```

// Input-2 Plot-2 Main Function

```
// main function which calls the respective algorithm for sorting.
int main(){
  // initializing variables and arrays
```

```
  vector< int > input ;
  int size , j ;
  // asking user for input of the number of elements
  cout << "Give the number of elements you would like the randomly generated input to have." << endl ;
  cin >> size ;
  // time variables
  clock_t begin , end ;
  // pushing size number of random values in range [ 1 , size ]
  for( j = 0 ; j < size ; ++j )
    input.push_back( rand( ) % ( size ) + 1 ) ;
  sort( input.begin( ) , input.end( ) ) ;
  // cross checking sorted array
  //cout << "Non-decreasing Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  insertion_sort( input , size ) ;
  end = clock( ) ;// calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for insertion sort is " << double( end - begin ) << " microseconds" << endl ;
  // cross checking random arrays
  //cout << "Non-decreasing Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  merge_print( input , 0 , size - 1 ) ;
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for merge sort is " << double( end - begin ) << " microseconds" << endl ;
  // cross checking random arrays
  //cout << "Non-decreasing Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  quick_print( input , 0 , size - 1 ) ;
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for deterministic quick sort is " << double( end - begin ) << " microseconds" <<
endl ;
}
```

// Input-3 Plot-3 Main Function

```
// main function which calls the respective algorithm for sorting.
int main(){
  // initializing variables and arrays
  vector< int > input ;
```

```
  int size , j ;
  // asking user for input of the number of elements
  cout << "Give the number of elements you would like the randomly generated input to have." << endl ;
  cin >> size ;
  // time variables
  clock_t begin , end ;
  // pushing size number of random values in range [ 1 , size ]
  for( j = 0 ; j < size ; ++j )
    input.push_back( rand( ) % ( size ) + 1 ) ;
  sort( input.begin( ) , input.end( ) , greater< int >( ) ) ;
  // cross checking sorted array
  //cout << "Non-increasing Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  insertion_sort( input , size ) ;
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for insertion sort is " << double( end - begin ) << " microseconds" << endl ;
  // cross checking sorted array
  //cout << "Non-increasing Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  merge_print( input , 0 , size - 1 ) ;
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time merge sort is " << double( end - begin ) << " microseconds" << endl ;
  // cross checking sorted array
  //cout << "Non-increasing Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  quick_print( input , 0 , size - 1 ) ;
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time deterministic quick sort is " << double( end - begin ) << " microseconds" << endl ;
}
```

// Input-4 Plot-4 Main Function

```
// main function which calls the respective algorithm for sorting.
int main(){
  // initializing variables and arrays
  vector< int > input ;
  int size , j ;
```

```
  // asking user for input of the number of elements
  cout << "Give the number of elements you would like the randomly generated input to have." << endl ;
  cin >> size ;
  // initializing variables with values
  double sum_ins = 0.0 ;
  double sum_merge = 0.0 ;
  double sum_quick = 0.0 ;
  // time variables
  clock_t begin , end ;
  for( int i = 0 ; i < 3 ; ++i ){
    // clearing input vector from the previous values stored so that a new set of random values can be added and later sorted to make 3 random sortings distinct
    // pushing size number of random values in range [ 1 , size ]
    input.clear( ) ;
    for( j = 0 ; j < size ; ++j )
      input.push_back( rand( ) % ( size ) + 1 ) ;
    sort( input.begin( ) , input.end( ) ) ;
    for( j = 0 ; j < 50 ; ++j )
      swap( &input[ rand( ) % size ] , &input[ rand( ) % size ] ) ;
    // cross checking random arrays
    //cout << "Random Array" << endl ;
    //print_array( input ) ;
    begin = clock( ) ;
    // here & operator isn't used so the sorted array won't be saved at the address we are accessing from input[ ] in the main function
    insertion_sort( input , size ) ;
    end = clock( ) ;
    // calculating runtime of sorting and adding it to the sum of runtimes
    sum_ins += double( end - begin ) ;
    cout << "Execution Time for " << ( i + 1 ) << " time for insertion sort is " << double( end - begin ) << " microseconds" << endl ;
    // cross checking random arrays
    //cout << "Random Array" << endl ;
    //print_array( input ) ;
    begin = clock( ) ;
    // here & operator isn't used so the sorted array won't be saved at the address we are accessing from input[ ] in the main functionselection_sort( input , size ) ;
    merge_print( input , 0 , size - 1 ) ;
    end = clock( ) ;
    // calculating runtime of sorting and adding it to the sum of runtimes
    sum_merge += double( end - begin ) ;
    cout << "Execution Time for " << ( i + 1 ) << " time for merge sort is " << double( end - begin ) << " microseconds" << endl ;
    // cross checking random arrays
    //cout << "Random Array" << endl ;
    //print_array( input ) ;
    begin = clock( ) ;
```

```
   // here & operator isn't used so the sorted array won't be saved at the address we are accessing from input[ ] in the main functionselection_sort( input , size ) ;
    quick_print( input , 0 , size - 1 ) ;
    end = clock( ) ;
    // calculating runtime of sorting and adding it to the sum of runtimes
    sum_quick += double( end - begin ) ;
    cout << "Execution Time for " << ( i + 1 ) << " time for deterministic quick sort is " << double( end - begin ) << " microseconds" << endl ;
  }
  // printing average of the runtimes of insertion and selection sort
  cout << "Average runtime for all 3 sortings of insertion sort is " << ( sum_ins / 3 ) << " microseconds" << endl ;
  cout << "Average runtime for all 3 sortings of merge sort is " << ( sum_merge / 3 ) << " microseconds" << endl ;
  cout << "Average runtime for all 3 sortings of deterministic quick sort is " << ( sum_quick / 3 ) << " microseconds" << endl ;
}
```

// Input-5 Main Function

```
// main function which calls the respective algorithm for sorting.
int main()
{
  // initializing variables and arrays
  vector< int > input ;
  int size = 100000 ;
  // pushing size number of random values in range [ 1 , size ]
  for( int i = 0 ; i < size ; ++i )
    input.push_back( rand( ) % ( 50 ) + 1 ) ;
  // time variables
  clock_t begin , end ;
  // cross checking random arrays
  //cout << "Random Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  insertion_sort( input , size ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from input[ ] in the main function
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for insertion sort is " << double( end - begin ) << " microseconds" << endl ;
  // cross checking random arrays
  //cout << "Random Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from input[ ] in the main function
  merge_print( input , 0 , size - 1 ) ;
  end = clock( ) ;
```

```
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for merge sort is " << double( end - begin ) << " microseconds" << endl ;
  // cross checking random arrays
  //cout << "Random Array" << endl ;
  //print_array( input ) ;
  begin = clock( ) ;
  // here & operator isn't used so the sorted array won't be saved at the address we are accessing from
input[ ] in the main function
  quick_print( input , 0 , size - 1 ) ;
  end = clock( ) ;
  // calculating runtime of sorting and adding it to the sum of runtimes
  cout << "Execution Time for deterministic quick sort is " << double( end - begin ) << " microseconds" <<
endl ;
}
```