Q1)

```
void mergesort(int data[], int first, int last)
{
    int mid = (first + last) / 2;//2, 1, 0, 4
    if (first ⩾ last)
        return;                    // base case: size = 1
    mergesort(data, first, mid);   // recursion: divide the list into halves
    mergesort(data, mid + 1, last);// recursion: divide the list into halves
    merge(data, first, mid, last); // start merging the list: conquer
}
int main ()
{
        data[]={9,0,1,2,8,10};
        mergesort(data, 0, 5);
        return 0;
}
```

The following functions are called by the following order:

```
mergesort(data, 0, 5)
        mergesort(data, 0, 2)
                mergesort(data, 0, 1)
                        mergesort(data, 0, 0)
                        mergesort(data, 1, 1)
                        merge(data, 0, 0, 1)
                mergesort(data, 2, 2)
                merge(data, 0, 1, 2)
        mergesort(data, 3, 5)
                mergesort(data, 3, 4)
                        mergesort(data, 3, 3)
                        mergesort(data, 4, 4)
                        merge(data, 3, 3, 4)
                mergesort(data, 5, 5)
                merge(data, 3, 4, 5)
        merge(data, 0, 2, 5)
```

Q3)
Running time comparison:

20:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 20

Quicksort of Integer data initially in random order
First five sorted elts:  5, 16, 21, 23, 25,
Last five sorted elts:  81, 86, 88, 92, 98,

The time use for quicksort is 2 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  5, 16, 21, 23, 25,
Last five sorted elts:  81, 86, 88, 92, 98,

The time use for mergeSort is 20 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  5, 16, 21, 23, 25,
Last five sorted elts:  81, 86, 88, 92, 98,

The time use for insertionSort is 2 microseconds
```

100:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 100

Quicksort of Integer data initially in random order
First five sorted elts:  0, 1, 1, 1, 1,
Last five sorted elts:  93, 95, 98, 98, 99,

The time use for quicksort is 16 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 1, 1, 1, 1,
Last five sorted elts:  93, 95, 98, 98, 99,

The time use for mergeSort is 42 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 1, 1, 1, 1,
Last five sorted elts:  93, 95, 98, 98, 99,

The time use for insertionSort is 17 microseconds
```

200:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 200

Quicksort of Integer data initially in random order
First five sorted elts:  0, 1, 2, 3, 4,
Last five sorted elts:  98, 99, 99, 99, 99,

The time use for quicksort is 35 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 1, 2, 3, 4,
Last five sorted elts:  98, 99, 99, 99, 99,

The time use for mergeSort is 96 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 1, 2, 3, 4,
Last five sorted elts:  98, 99, 99, 99, 99,

The time use for insertionSort is 57 microseconds
```

400:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 400

Quicksort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 1,
Last five sorted elts:  98, 98, 99, 99, 99,

The time use for quicksort is 75 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 1,
Last five sorted elts:  98, 98, 99, 99, 99,

The time use for mergeSort is 165 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 1,
Last five sorted elts:  98, 98, 99, 99, 99,

The time use for insertionSort is 386 microseconds
```

800:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 800

Quicksort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 1, 1,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for quicksort is 148 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 1, 1,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for mergeSort is 290 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 1, 1,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for insertionSort is 826 microseconds
```

1600:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 1600

Quicksort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for quicksort is 426 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for mergeSort is 556 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for insertionSort is 3752 microseconds
```

3200:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 3200

Quicksort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for quicksort is 703 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for mergeSort is 1131 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for insertionSort is 18091 microseconds
```

6400:

```
 ----- quicksort/mergesort/insertion comparison -----

How many elements to sort: - 6400

Quicksort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for quicksort is 1644 microseconds

Mergesort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for mergeSort is 2274 microseconds

Insertion sort of Integer data initially in random order
First five sorted elts:  0, 0, 0, 0, 0,
Last five sorted elts:  99, 99, 99, 99, 99,

The time use for insertionSort is 49593 microseconds
```

Q4) DanceSort (Dsort)

Description:

A group of dancers represents individual elements in an unsorted array. Each dancer has a number assigned to them. Start from first dancer of array(pivot), compare their number with the rest of the dancers, and swap places with dancers having a smaller value. This is symbolized by transferring their hats. Then pivot continues to compare himself to dancers yet to be compared, until the adjacent dancers on both sides have been compared. Then pivot takes its current position and is hence sorted. For every sorted element, the array splits into 2 sub-arrays. Then the process repeats for each sub-array.

Comparison:

The difference between the DanceSort (Dsort) and the quicksort algorithm in lecture lies within the division of array into high and low. As well as selection of pivot. Dsort always starts with the fist element of array/subarray as pivot, whilst in the lecture random elements can be chosen as pivot.

Dsort has two pointer, "pivot" and "hat". Iteratively compare data[pivot] with data[hat]. If data[hat] < data[pivot] → swap. Now iterate a second time. If data[hat]>data[pivot] → swap and return to first loop. Continue until data[pivot]=data[hat]. All elements perform comparison with pointer.

Whereas in the lecture notes, a random element can be chosen as pivot. The subarrays are first recursively sorted by iterating through each element from end of array (high). If high is smaller than element[pivot+1] (low), then low swap with high. Continue iteration until low>=high, and exchange pivot with high.

Pseudocode:

```
Dsort(data, p, h)
{
  if (p ⩾ h) return;
  int pivot = p;
  int hat = h;
  while (true)
  {
          if hat > pivot
          {
          while (data[hat] ⩾ data[pivot] and hat > pivot)
                  hat--;
          swap(data[hat], data[pivot]);
          swap(hat, pivot)
          }
          if hat < pivot
      {
      while (data[hat] ⩽ data[pivot] and hat < pivot)
```

```
            hat++;
          swap(data[hat], data[pivot]);
      swap(hat, pivot)
      }
      if hat == pivot
      break out of the while loop
  }
}
```