1. Sorting

3.1 Soring Basic Concept

a. In place sorting

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

b. Internal and external sorting

When all data that needs to be sorted cannot be placed in-memory at a time, the sorting is called external sorting. External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting. Some extrenal storage like hard-disk, CD, etc is used for external storage.

When all data is placed in-memory, then sorting is called internal sorting.

c. Stable sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

3.2 Selection Sort

a. **Algorithm**

The algorithm **keeps selecting the min/max** in a given array.

It maintains two subarrays in a given array. One subarray is already sorted, the remaining subarray is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is **SELECTED** and moved to the sorted subarray.

All the elements in the unsorted array are greater(if sorted in ascending order) than the sorted array. This is because the sorted array is first k min elements in the given array.

b. C++ implementation

```
void selectionSort(int arr[], int n)
{
   int i, j, min_idx;

  // One by one move boundary of unsorted subarray
  for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
      {  if (arr[j] < arr[min_idx])
         min_idx = j;
       }
      // Swap the found minimum element with the first element
      swap(&arr[min_idx], &arr[i]);
   }
}
```

c. Complexity

Time complexity:

Average: O(n^2) comparisons, O(n) swaps
Best: O(n^2) comparisons, O(n) swaps
Worst: O(n^2) comparisons, O(n) swaps

Space complexity:

O(1)

The good thing about selection sort is it never makes more than O(n) swaps/write and can be useful when memory write is a costly operation.

3.3 Insertion Sort

a. Algorithm

// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
    Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

b. C++ implementation

```
void insertionSort(int arr[], int n)
{
        int i, key, j;
        for (i = 1; i < n; i++)
        {
        Key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
                arr[j+1] = arr[j];
                j = j-1;
        }
        arr[j+1] = key;
        }
}
```

c. Complexity

Time complexity:

Worst O(n^2) comparisons and swaps: Elements are reversely sorted.
Best O(n) comparisons and 0 swaps: Elements are already sorted.

Space complexity:

O(1)

3.4 Bubble sort

a. Algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

b. c++ implementation

```
void bubbleSort(int arr[], int n)
{
  int i, j;
  bool swapped;
  for (i = 0; i < n-1; i++)
  {
    swapped = false;
    for (j = 0; j < n-i-1; j++)
    {
      if (arr[j] > arr[j+1])
      {
        swap(&arr[j], &arr[j+1]);
        swapped = true;
      }
    }

    // IF no two elements were swapped by inner loop, then break
    if (swapped == false)
      break;
  }
}
```

3.5 QuickSort

a. Algorithm

1) Randomly choose a pivot element
2) partition

   Given an array and pivot element, put x at its correct position in sorted array such that all elements smaller than x before x, and put all elements greater than x after x.

   This is done by keep track of an index i which rightmost elements that are smaller than the pivot. For each jth element, if arr[j] < pivot

   Then swap a[j] with a[i+1], i++

3) keep doing 1) and 2) in the partitioned array

b. implement in c++

```
int partition (int arr[], int low, int high)
{
   int pivot = arr[high];    // pivot
   int i = (low - 1);  // Index of smaller element

   for (int j = low; j <= high- 1; j++)
   {
      // If current element is smaller than or
      // equal to pivot
      if (arr[j] <= pivot)
      {
         i++;    // increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

 void quickSort(int arr[], int low, int high)
{
 if (low < high)
 {
```

```
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
        }
    }
```

c. Time complexity

Average case: nlogn

Worst case(with a sorted array and bad pivot): n^2, the algorithm goes to insertion sort

d. Space complexity: log n

Since it is a recursion call and it needs a stack frame to store function address. The length of the stack frame is log n

e. Quick sort is in-place but not stable.

f. Comparison with heap sort

1) worst time: heap sort is better as it guarantees n logn
2) average time: quicksort is better as the prefactor of n logn is smaller

g. Comparison with merge sort

1) Merge sort is a stable sort, quick sort is not.
2) Space. Merge sort requires O(n) space.

3.6 Merge Sort

a. Algorithm

MergeSort(arr[], l, r)
If r > l

1) Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2) Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3) Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4) Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

b. Implementation in c++

```
// Merges two subarrays of arr[]. First subarray is arr[l..m]
Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
```

```
    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
```

```
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

c. **Complexity:**
   Time complexity:
   Average: nlog(n)
   Best: nlog(n)
   Worst: nlog(n)
   Space complexity:
   Depending on the give data structure.
   If it is an array then it needs n additional spaces, however if it is linkedlist, then it does not need additional space
d. **Not in-place but stable**
e. **Application**
   1) Sort linkedlist.
      Because merge two linkedlists can be done by inserting element from one list to the other, and inserting element in the linkedlist needs O(1) in space, and O(1) in time. Therefore merge sort is useful for sorting linked list.
   2) External sorting

3.7 External Merge Sorting for large amount of data example

Since merge sort is a divide and conquer algorithm, it can be used to sort large amount of data. The algorithm first sorts *M* items at a time and puts the sorted lists back into external memory. It then recursively does a M/B merge on those sorted lists. To do this merge, *B* elements from each sorted list are loaded into internal memory, and the minimum is repeatedly outputted.

Example, sort 900 megabytes of data using only 100 megabytes of RAM:

1) Read 100 MB of the data in main memory and sort by some conventional method, like quicksort.
2) Write the sorted data to disk.
3) Repeat steps 1 and 2 until all of the data is in sorted 100 MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.
4) Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)
5) Perform a 9-way merge and store the result in the output buffer. Whenever the output buffer fills, write it to the final sorted file and empty it. Whenever any of the 9 input buffers empties, fill it with the next 10 MB of its associated 100 MB sorted chunk until no more data

from the chunk is available. This is the key step that makes external merge sort work externally -- because the merge algorithm only makes one pass sequentially through each of the chunks, each chunk does not have to be loaded completely; rather, sequential parts of the chunk can be loaded as needed.