INSERTION SORT

Space Complexity
$O(1)$
Time Complexity
$O(n^2)$

```
void insertion_sort(int arr[], int n){
    int i, key, j;
    for(i=1; i<n; i++){
        key=arr[i];        // select the first unsorted
                           element.
        j=i-1;

        while (j>=0 && arr[j] > key){
            arr[j+1] = arr[j];
            j=j-1;         this loop shifts all
                           the elements to right
                           to create the position
                           for unsorted element.
        } // end of while
        arr[j+1] = key;
    } // end of for        This inserts the
                           unsorted element to
} // end of insertion_sort its correct position
```

SELECTION SORT

Time $O(n^2)$
Space $O(1)$

```
void selection_sort (int arr[], int n){
    int i, j, min_idx;
                                    0 to (n-2)
    for (i=0; i<n-1; i++){          // n-2 passes
        min_idx = i;
        for (j=i+1; j<n; j++){
            if (arr[j] < arr[min_idx])
                min_idx = j;        // update the
        } // end of inner for          index
        swap(&arr[min_idx], &arr[i])   // swapping is
                                       outside inner
                                       loop
    } // end of outer for
} // end of selection sort
```

Bubble Sort

```
void bubblesort(int arr[], int n){
    int i,j;
    for(i=0; i<n-1; i++){
        for(j=0; j<n-i-1; j++){
            if(a[j] > a[j+1]){
            # swap(a[j], a[j+1]) #
            } // endif
        } // inner for
    } // end of outterfor
} //end function bubble_sort
```

Quick SORT

```
int partition(int a[], int l, int h){
    int pivot = a[h];
    int i = l-1;
    for(int j = l; j<=h-1; j++){
        if (a[j] <=pivot){
            i++;
            swap (a[i],a[j])
        } //endif
    } // end of ~~inner~~ for
    swap (a[i+1], a[h]) ;//outside
    return(i+1);     // place pivot
                     to its correct
                     position
}
```

<2>

<1>

```
void quicksort (int a[], int l
                int h){
    if(l<h){
        int pi=partition(a,l,h);

        quicksort(a,l,pi-1);
        quicksort (a,pi+1,h);
```

HEAP SORT

```
void heap-sort (int a[], int n){
  // build heap (rearrange array
  for(int i=n/2-1; i>=0; i--)
      heapify (arr,n,i);          ———— Creates a max heap

  // one by one extract an element from heap
  for(int i=n-1; i>=0; i--){
move current root to the end > swap (arr[0], a[i]);   ———— swap first and the last node
      heapify (arr,i,0);          ———— Creates max heap and reduced the Array (reduced heap)
  }

void heapify (int a[], int n, int i){
// To heapify a subtree rooted with node i which is
// an index in a[], n is size of heap
    int largest = i;      // initialize largest as root
    int l = 2*i+1;        // left child.
    int r = 2*i+2;        // right child
    if (l<n && a[l] > a[largest])
        largest = l;            // if left child is larger than root
    if (r<n && a[r] > a[largest])
        largest = r;            // if right child is larger than root
    // if largest is not root
    if (largest != i){
        swap (a[i], a[largest]);
        heapify (a, n, largest);     // recursively heapify the affected sub-tree
    } // end of if
} // end of heapify
```

```
int getMax(int a[], int n){
    int mx = a[0], i;
    for(i=0; i<n; i++){
        if(a[i]>mx){
            mx = a[i];
        }
    }
    return mx;          // get maximum number from a[]
}
```

RADIX SORT

```
        //a function to do counting sort of a[] according to
        the digit represented by exp.
void countSort(int a[], int n, int exp){
    int op[n];
    int i, count[10] = {0};

    for(i=0; i<n; i++)                        // store count of
F1:     count[(a[i]/exp)%10]++;               occurrences in count[]

    for(i=1; i<10; i++)                       // change count[i] so that
F2:     count[i] += count[i-1];               count[i] now contains
                                              actual position of this
                                              digit in output
Build the output Array
    for(i=n-1; i>=0; i--){
F3:     op[count[(a[i]/exp)%10]-1] = arr[i];
        count[(a[i]/exp)%10]--;
    }

    for(i=0; i<n; i++)                        // copy the output array to
F4:     a[i] = op[i];                         a[], so that a[] contains
    }                                         sorted numbers according
                                              to current digit
                                              exp
```

RADIX SORT

```
void radixSort (int a[], int n){
    int m= getMax (a, n);
    for (int exp=1; m/exp>0; exp*=10)
        countSort (a, n, exp);
}
```

Sort based on the digit exp 10^i

MERGE SORT

```
void merge (int a[], int l, int m, int r){
    int i, j, k;
    int n1 = m-l+1;
    int n2 = r-m;

    int L[n1], R[n2];
    int temp[50];
    for (i=0; i<n1; i++)
        L[i] = arr[l+i];

    for (j=0; j<n2; j++)
        R[j] = arr[m+1+j];

    i=0;  // initialize index of first subarray
    j=0;  // initial index of second subarray
    k=l;  // initial index of merged subarray
    while (i<n1 && j<n2){

        if (a[i]<=a[j])
            temp[k++] =a[i++];

        else
            temp[k++] =a[j++];

    } //end of while
```

create temporary Arrays

Copy data to the left & right temporary arrays L[·] & R[·] respectively

MERGE SORT

```
while (i < n1){            // or i <= m , copy the remaining
                           //   elements of the first
                           //   (left) list
    temp[k++] = a[i++];
}

while (j < n2) {           // or, j <= r , copy the remaining elements
                           //   of the second list
                           //   (right)
    temp[k++] = a[j++];
}
                           // Transfer element from temp[.] to
                           //   a[]
for (i = l, j = 0; i <= r ; i++, j++)
    a[i] = temp[j];

}

void mergesort(int a[], int l, int r){
    int mid;
    if (l < r){
mid = l+(r-l)/2  or,   mid = (l+r)/2;
        mergesort(a, l, m);          // left recursion
        mergesort (a, m+1, r);       // right recursion

        merge(a, l, m, r);

    }
    else{
        return;                      // single element in the
                                     //   subarray
                                     //   (sorted)
    }

} // end of mergesort
```