QUICK SORT

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo        // place for swapping
    for j := lo to hi - 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Worst case performance	$O(n^2)$
Best case performance	$O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys)
Average case performance	$O(n \log n)$

INSERTION SORT

```
for i = 1 to length(A) - 1
    x = A[i]
    j = i - 1
    while j >= 0 and A[j] > x
        A[j+1] = A[j]
        j = j - 1
    end while
    A[j+1] = x[3]
 end for
```

Worst case performance	$O(n^2)$ comparisons, swaps
Best case performance	$O(n)$ comparisons, $O(1)$ swaps
Average case performance	$O(n^2)$ comparisons, swaps

HEAP SORT

```
void maxHeapify(long a[], int i, int size)
{
    int l=leftChild(a,i),r=rightChild(a,i),largest;
    if(l<=size-1 && a[l]>a[i])
        largest =l;
    else largest =i;
    if (r<=size-1 && a[r]>a[largest])
        largest=r;
    if(largest!=i)
    {
        swap(&a[i],&a[largest]);
        maxHeapify(a,largest,size);
    }
}

void buildMaxHeap(long a[],int size)
{
    int i;
    for(i=(size-1)/2;i>=0;i--)
        maxHeapify(a,i,size);
}

void heapSort(long a[],int *size)
{
    buildMaxHeap(a,*size);
    int i;
    for(i=*size-1;i>=1;i--)
        {
            swap(&a[0],&a[i]);
            (*size)--;
            maxHeapify(a,0,*size);
        }
}
```

Worst case performance	$O(n \log n)$
Best case performance	$\Omega(n), O(n \log n)$ [1]
Average case performance	$O(n \log n)$
Worst case space complexity	$O(1)$ auxiliary

MERGE SORT

```
i = 0;
j = 0;
k = start;
while (i < n1 && j < n2)
{
      if (Leftarray[i] <= Rightarray[j])
      {
          arr[k] = Leftarray[i];
          i++;
      }
      else
      {
          arr[k] = Rightarray[j];
          j++;
      }
      k++;
}

while (j < n2)
{
         arr[k++] = Rightarray[j++];
}
while(i<n1)
{
            arr[k++] = Leftarray[i++];
}
```

Worst case performance	O(n log n)
Best case performance	O(n log n) typical, O(n) natural variant
Average case performance	O(n log n)
Worst case space complexity	O(n) total, O(n) auxiliary

SELECTION SORT

```
for(i=0;i<size;i++)
{
      smallest=i;
      for(j=i+1;j<size;j++)
      {

      if(strcmp(x[smallest],x[j])>0)
            {
                  smallest=j;
            }
      }
      swapStrings(x[i],x[smallest]);
}
```

Worst case performance	O(n^2)
Best case performance	O(n^2)
Average case performance	O(n^2)

BUBBLE SORT

```
void bubbleSort(float arr[],int size)
{
    if(size==1)
        return;
    int i,j;
    for(i=0;i<size-1;i++)
        if(arr[i]>arr[i+1])
            swap(&arr[i],&arr[i+1]);
    bubbleSort(arr,size-1);
}
```

Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$
Worst case space complexity	$O(1)$ auxiliary