MODIFIED BUBBLE SORT

1. Accept n, the number of elements
2. Accept the data (into info)
3. sorted ← false
4. Loop pass number from 1 to n
   4.1 If sorted = false
      4.1.1 sorted ← true
   4.2 Loop i from 1 to n – pass number
      4.2.1If i^{th} element < $(i+1)^{th}$ element
         4.2.1.1Switch them
         4.2.1.2 sorted ← false
      4.2.2 else STOP
5. Display the list info

Best Case:

Assuming list is already in ascending order.

Outer loop will execute only once & n – 1 comparisons will be made by inner loop.

Hence efficiency is of the order of n i.e. O(n).

Worst Case:

Assuming list is in descending order.

The outer loop will execute n – 1 time

Each time the inner loop runs n – 1 times, n – 2 times, n – 3 times,…… , 1 time

Hence,

Pass = 0 comparisons = n – 1

Pass = 1 comparisons = n – 2

Pass = 2 comparisons = n – 3

……

The sum of comparisons is an Arithmetic Progression and hence the order in n^2 i.e. $O(n^2)$.

INSERTION SORT

1. Accept n, the number of elements
2. Accept the data (numbers) into info
3. Loop pos from 2 to n in steps of 1
   3.1 y←info(pos)
   3.2 Loop i from pos -1 to 1st element steps of -1
      3.2.1 if y < info(i)
         3.2.1.1 x(i+1)← x(i)
      3.2.2 else Break out of loop (goto 3.3)
   3.3 x(i+1) ← y
4. Display the sorted array, info

Best Case:

Assuming list is already in ascending order.
Outer loop will execute n – 1 times. Inner loop will execute only once per outer loop cycle.
Hence,
Pos = 2 comparisons = 1
Pos = 3 comparisons = 1
……
Sum of comparisons = n – 1
Hence efficiency is of the order of n i.e. O(n).

Worst Case:
Assuming list is in descending order.
The outer loop will execute n – 1 time
Each time the inner loop runs 1 time, 2 times, 3 times,…n – 1 times
Hence,
Pos = 2 comparisons = 1
Pos = 3 comparisons = 2
Pos = 4 comparisons = 3
……
The sum of comparisons is an Arithmetic Progression and hence the order in n^2 i.e. $O(n^2)$.

SHELL SORT

1. Accept n, the number of elements
2. Accept the data (numbers) into info
3. Set span values (say, 5, 3, 1)
4. Select the next span value
   4.1 Loop pos from span to n in steps of 1
      4.1.1 y ← info(pos)
      4.1.2 Loop i from pos – span to 1st element in steps of –span
         4.1.2.1 If y<x[i]
            4.1.2.1.1 x(i+span) ← x(i)
         4.1.2.2 else go to step 4.1.3
      4.1.3 x(i+span) ← y
5. Repeat step 4 till all span values have been tried
6. Display sorted list

At worst O(n3/2). It's also possible to use sequences other than 1, 4, 13, ... but they do not necessarily give the same efficiency.

RADIX SORT

1. Accept n, the number of elements
2. Accept the numbers into array arr
3. Loop j from units place till no more digits can be extracted
   3.1 Reset all 10 queues by setting front & rear to -1
   3.2 Loop through all n numbers
   3.3 Extract the j^{th} digit
   3.4 Insert the number into j^{th} queue
   3.5 Extract all numbers from the queues
4. Display the sorted list

If there are n numbers & the maximum size of a number is m digits, then in each pass we extract a digit & insert into a specific queue. We have to do this for all n numbers.

Thus, for units digit n times the above procedure is repeated

For tenths digit again we repeat the procedure n times

For hundredths place we do the same n times

…….

Since the maximum size of the number is m hence the above goes on m times. Hence the sum is n + n + …. m times=mn
Thus efficiency is O(mn)

MERGE SORT

1. Accept n, the number of elements
2. Accept the data into array info
3. size ← 1
4. Loop while size < n
   - 4.1 l1 ←0
   - 4.2 k ← 0
   - 4.3 loop while l1+size<n i.e. while l2 can be found
     - 4.3.1 l2 ← l1 + size
     - 4.3.2 u1 ← l2 -1
     - 4.3.3 u2 ← l2 + size – 1 if it exists else u2 ← n -1
     - 4.3.4 i ← l1
     - 4.3.5 j ← l2
     - 4.3.6 Loop till i ≤ u1 and j ≤ u2
       - 4.3.6.1 if info(i) ≤ info(j)
         - 4.3.6.1.1 aux(k) ← x(i)
         - 4.3.6.1.2 k ← k+1
         - 4.3.6.1.3 i← i+1
       - 4.3.6.2 else
         - 4.3.6.2.1 aux(k) ← x(j)
         - 4.3.6.2.2 k ← k+1
         - 4.3.6.2.3 j ← j+1
     - 4.3.7 Dump x(i) into aux(k) if i ≤ u1
     - 4.3.8 Dump x(j) into aux(k) if j ≤ u2
     - 4.3.9 l1 ← u2 + 1
     - 4.3.10 Dump remaining elements from x(i) to aux(k)
     - 4.3.11 Copy aux into x
     - 4.3.12 Size ← size * 2
5. display sorted array x

We have first a partition size of 1 and having total n comparisons.

Then partition size is 2 with n comparisons again

Then partition size is 4 with n comparisons

This goes on till we don't have any partitions available.

If there are n numbers we have $\log_2 n$ passes, with n comparisons in each pass.

Thus the efficiency is of the order of O(n log n).

QUICK SORT

1. Accept n, the number of elements
2. Accept the data into array info
3. l ← 1, r←n
4. i ← l, j←r
5. pivot ← a(l)
6. Loop while i<j
   6.1. loop while a(i) < pivot
      6.1.1. i ← i+1
   6.2. loop while a(j)>pivot
      6.2.1. j ← j -1
   6.3. If i<j
      6.3.1. Swap a(i) and a(j)
7. If l<j
   7.1. Repeat step 4 onwards taking r ← j -1
8. If i<r
   8.1. Repeat step 4 onwards taking l ← i + 1

Assuming that n is 2^m.

Best Case:

In the first pass we have n comparisons.

Then assuming we have 2 partitions with approx. n/2 elements in each, giving further n comparisons (approx.) This goes on till we are left with a partition of size 1.

Since we have assumed n is 2^m, thus the above procedure will go on m times & we would get a sum of approx m*n comparisons. But m = $\log_2 n$, thus total number of comparisons is of the order of n(log n) i.e. O(n log n).

Worst case:

If the 1st pass results only in one partition i.e. say only left or only right (this will happen when the pivot is either the largest or smallest in the list), then we have n comparisons in the 1st pass & then n – 1 in the 2nd pass & n – 3 in the 3rd pass & so on…. Thus we have n – 1 passes with number of comparisons in each pass being n, n -1, n – 2, n – 3, …….

Thus the sum of comparisons is an arithmetic progression and hence is order of n^2 i.e. $O(n^2)$.

SELECTION SORT

1. Accept n, the number of elements
2. Accept the data into array x
3. loop i from last position i.e. n -1 to 0 in steps of -1
   3.1 large ← x(0)
   3.2 indx ← 0
   3.3 loop j from 1 to i in steps of 1
      3.3.1 if x(j) > large
         3.3.1.1 update large ← x(j)
         3.3.1.2 update indx ← j
   3.4 x(indx) ← x(i)
   3.5 x(i) ← large
4. Display sorted list , i.e. x

Pass 1: n – 1 comparisons

Pass 2: n – 2 comparisons

Pass 3: n – 3 comparisons

……

Sum of comparisons is an arithmetic progression. Hence the efficiency is of the order of n^2 i.e. $O(n^2)$

LINEAR SEARCH

1. Accept n, the number of elements
2. Accept the data into x
3. Accept the data to search, say target
4. Loop i from 0 to n -1 in steps of 1
   4.1 if x(i) = target
      4.1.1 Display data found
      4.1.2 Stop
5. Display no data found

Best Case:

The best case scenario is that we find the target in the first attempt i.e. at the beginning of the list. Hence the efficiency is O(1).

Worst case:

This is when we find the target at the end of the list or is absent in the list. In this case we go through the entire list (i.e. n comparisons). Hence the efficiency is O(n).

BINARY SEARCH

1. Accept n, the number of data items
2. Accept the data into array x
3. Accept target, the element to search
4. low ← 0
5. high ← n -1
6. loop while low ≤ high
   6.1 mid = (low+high)/2
   6.2 if target < x(mid)
      6.2.1 high = mid -1
   6.3 if target > x(mid)
      6.3.1 low = mid +1
   6.4 if target = x(mid)
      6.4.1 Display position mid
      6.4.2 Stop
7. Display not found

Best Case:

Again the best case scenario would be that we find the element in the first attempt i.e. at the centre. In this case the efficiency is of the order of O(1).

Worst Case:

This will be simulated when the data to be found (target) is not present in the list or will be the last one found after the partitioning leads us to only one element in the partition.

Each time we break up the list into half, hence the maximum number of partitions are $\log_2 n$. We make one comparison per pass (i.e. with mid only) & totally we have log n passes. Hence the efficiency in O(log n).