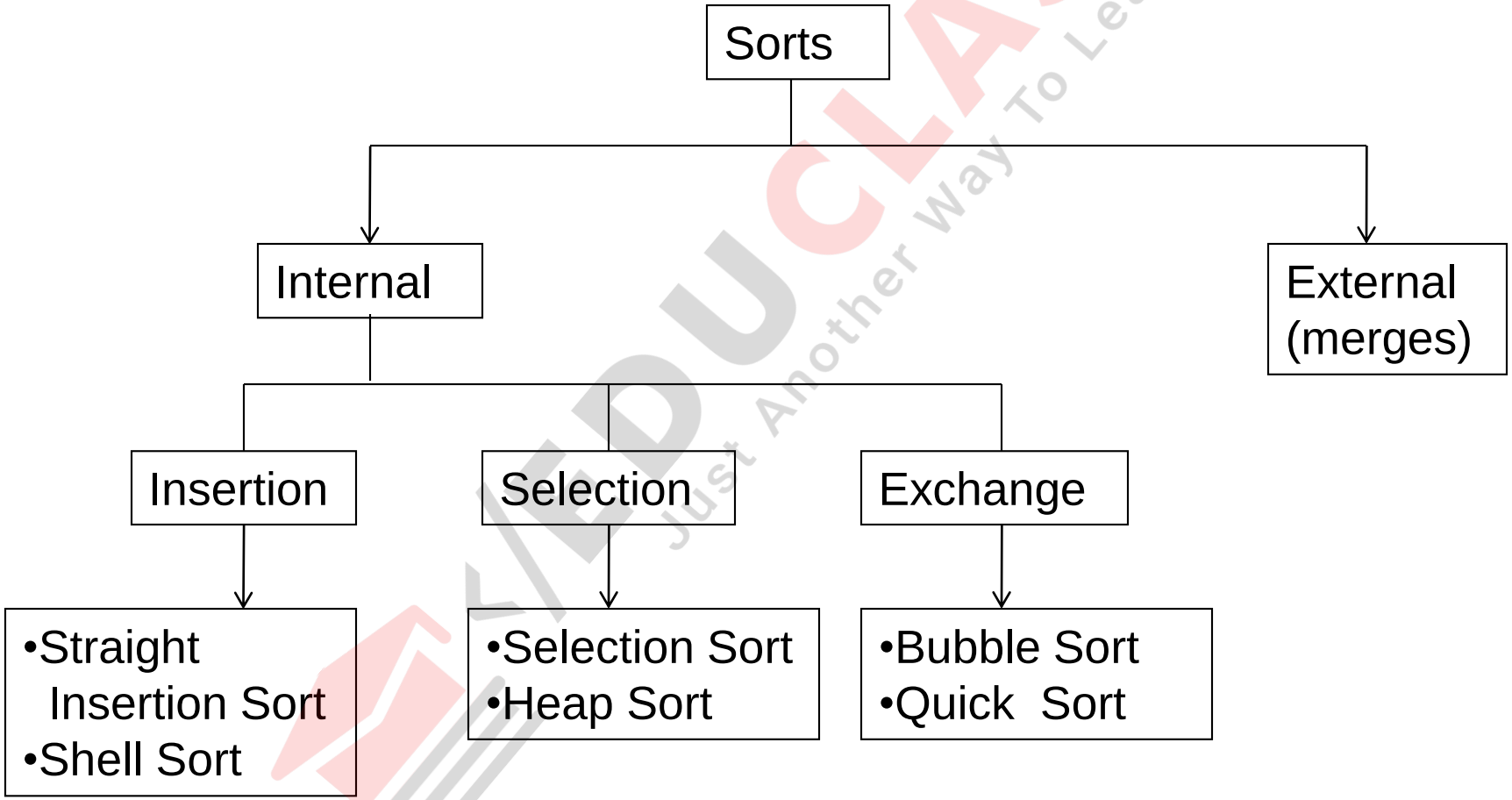
Sorting

Sorting Hierarchy



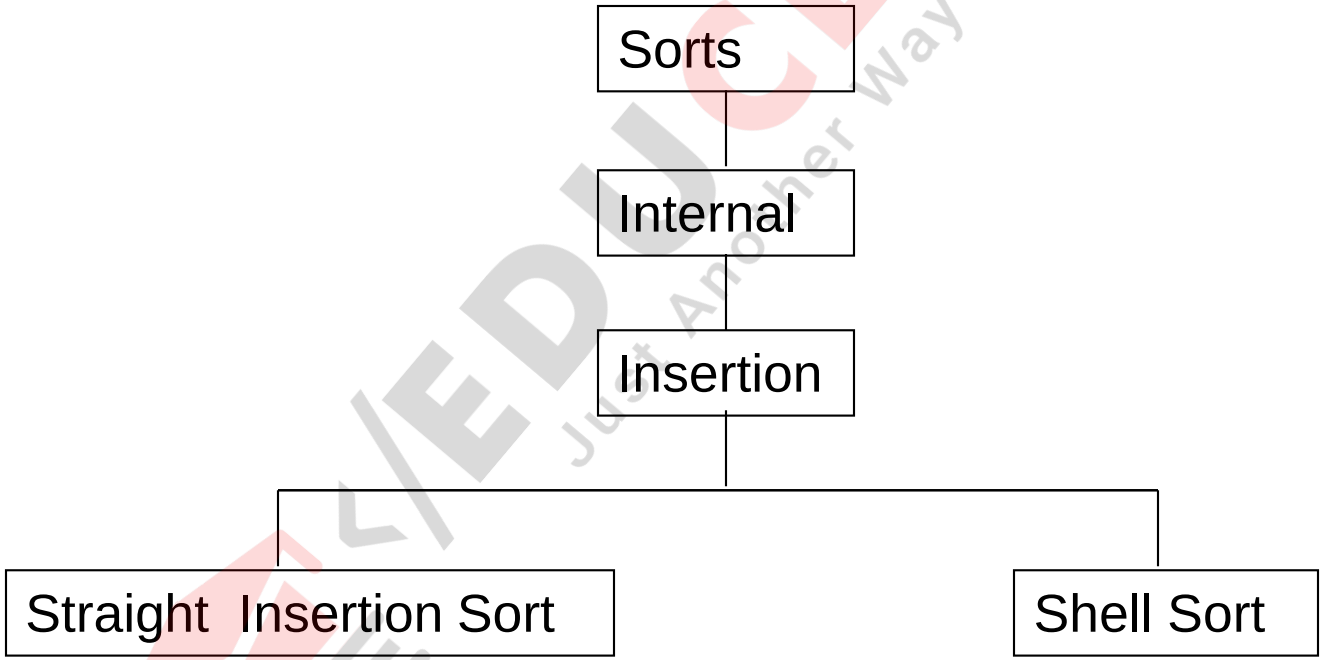
Difference between Internal Sort and External Sort

- Internal Sort:
  - All of the data are held in primary memory during the sorting process
- External Sort:
  - Uses primary memory for the data currently being sorted and secondary storage for any data that does not fit in primary memory.
  - Eg:

Sort Efficiency

- An estimate of the number of comparisions and moves required to order an unordered list.

Insertion Sorts



Straight Insertion sort

- The list is divided into 2 parts:
  - Sorted and Unsorted
- In this, the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.
- If there is a list of n elements , the straight insertion sort will take at most n-1 passes to sort the data.
- Algorithm

Algorithm insertionsort(int a[], int noofelements)

1. [Declare]

   curdata, prevdata,temp

2. For curdata=1 to to curdata <noofelements

   1.For prevdata=curdata to prevdata>0step -1

      1. if(x[prevdata-1]>x[prevdata])

         a.temp=x[prevdata]

         b. x[prevdata]=x[prevdata-1]

         c. x[prevdata-1]=temp

3. Display array x after sorting

Sort efficiency of Insertion Sort

- The outer loop executes n-1 times(from 1 to noofelements).
- The inner loop is dependant on the outer loop(prevdata=curdata to 1 step -1), we have a dependant quadratic loop.
- in big-O notation the efficiency of insertion sort is **$O(n^2)$**

Shell Sort

- Shell sort is named after its creator, Donald Shell.
- Improved version of straight insertion sort.
- This method is also called **diminishing-increment sort**
- **Complexity of shell sort :**
- **Worst case performance :depends on gap sequence. But the known is : $O(n\log_2 n)$**
- **Best case performance O(n)**
- **Average case performance :depends on gap sequence**
- Algorithm

```
void shell_sort(int  a[],int size)
1.[Declare]
    temp,gap,i,j,xchng=1
2.  For gap=size/2 to gap> 0 step gap/2
    1.do
       A.xchng=0
       B. for i=0 to i<size-gap
           1.  if(a[i]>a[i+gap])
                a. temp=a[i]
                b. a[i] =a[i+gap]
                c. a[i+gap]=temp
                d. xchng=1
      while(xchng ==1)
3. display_sorted_list(list,noofelements)
```

1. Size is divided by 2 in each loop. (gap=size/2 to gap> 0 step gap/2). Therefore, it is executed **logn** times.
2. The first inner loop executes size-gap times i.e
   (size-(size/2)) for the first time.
   (size-(size/4)) for the second time and so on…

Therefore, we have

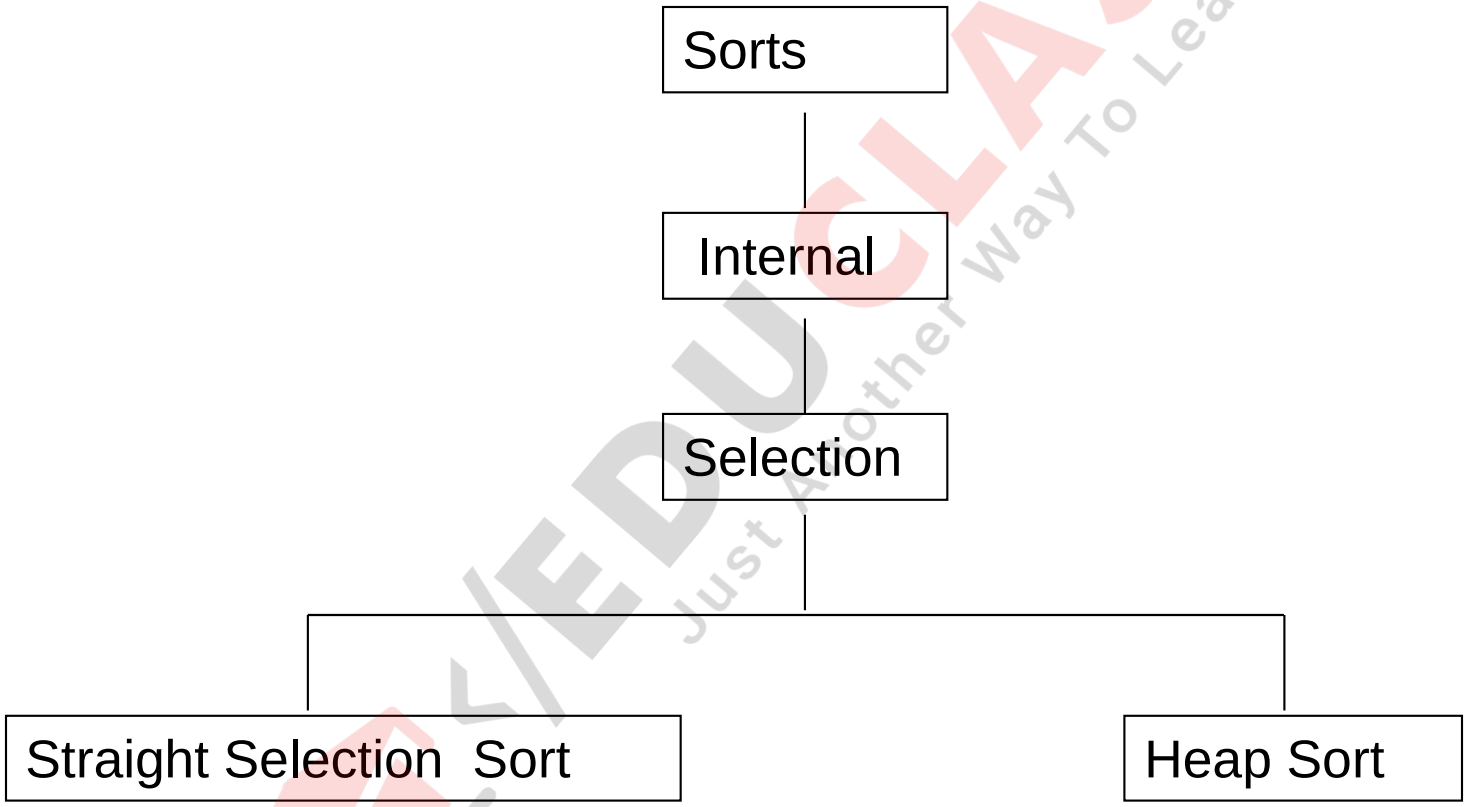
$\log_2 n$ *[(size-(size/2)) +
(size-(size/4)) +
(size-(size/8))…+1]

= $n\log_2 n$

3. In big-O notation the effciency of insertion sort is **$O(n\log_2 n)$**

- **Advantage of Shell sort:**
  - Its only efficient for medium size lists. For bigger lists, the algorithm is not the best choice.
- **Disadvantage of Shell sort:**
  - It is a complex algorithm and its not nearly as efficient as the merge, heap, and quick sorts.
  - The shell sort is still significantly slower than the merge, heap, and quick sorts.

Selection Sort



Straight Selection Sort

- In this sort, select the smallest item and place it in the sorted order.
- The list at any moment is divided into 2 sublists viz sorted and unsorted which are divided by an imaginary wall.
- Easiest method of sorting.

Straight Selection Sort

- In this , select the beginning element and the smallest element in the list and exchange. After each selection and exchange, the wall between the sorted sublist and unsorted sublist moves one element increasing the number of sorted elements and decreasing the number of unsorted elements.
- Each time one element is moved from unsorted sublist to the sorted sublist, one pass is completed.
- For n elements, we need n-1 passes to completely rearrange the data.

Algorithm of selection sort

- Diagram:
- Algorithm:

Algorithm selection(a[], noofelements)

Pre :list :array of integer elements

last:nth element in the list

Post:sorted list of data

1. For curdata = 0 to curdata< (noofelements-1)
    1. For nextdata = curdata +1 to nextdata<noofelements
        1. if a[curdata]>a[nextdata]
            swap the values
2. Print the sorted array

Efficiency of Selection Sort

- The outer loop executes n-1 times(from 0 to last -1).
- The inner loop is dependant on the outer loop(subsequentdata = currentdata +1),we have a dependant quadratic loop.
  - in big-O notation the efficiency of insertion sort is **$O(n^2)$**

Heap Sort

- The heap sort algorithm is an improved version of straight selection sort.
- Based on a tree structure that reflects a particular order of a corporate hierarchy.
- i.e. In the corporate management, president is at the top. When the president retires, the VP competes for the job and becomes the president and creates a vacancy.Hence the vacancy continuously appears at the top. This idea illustrates the heap sort method.

- The heap sort proceeds in two phases:
  - The entries are arranged in the heap(build_heap)
  - Remove the element from the top of the heap and promote another entry to take its place.

Heap sort algorithm

Heap_Sort(A,n)

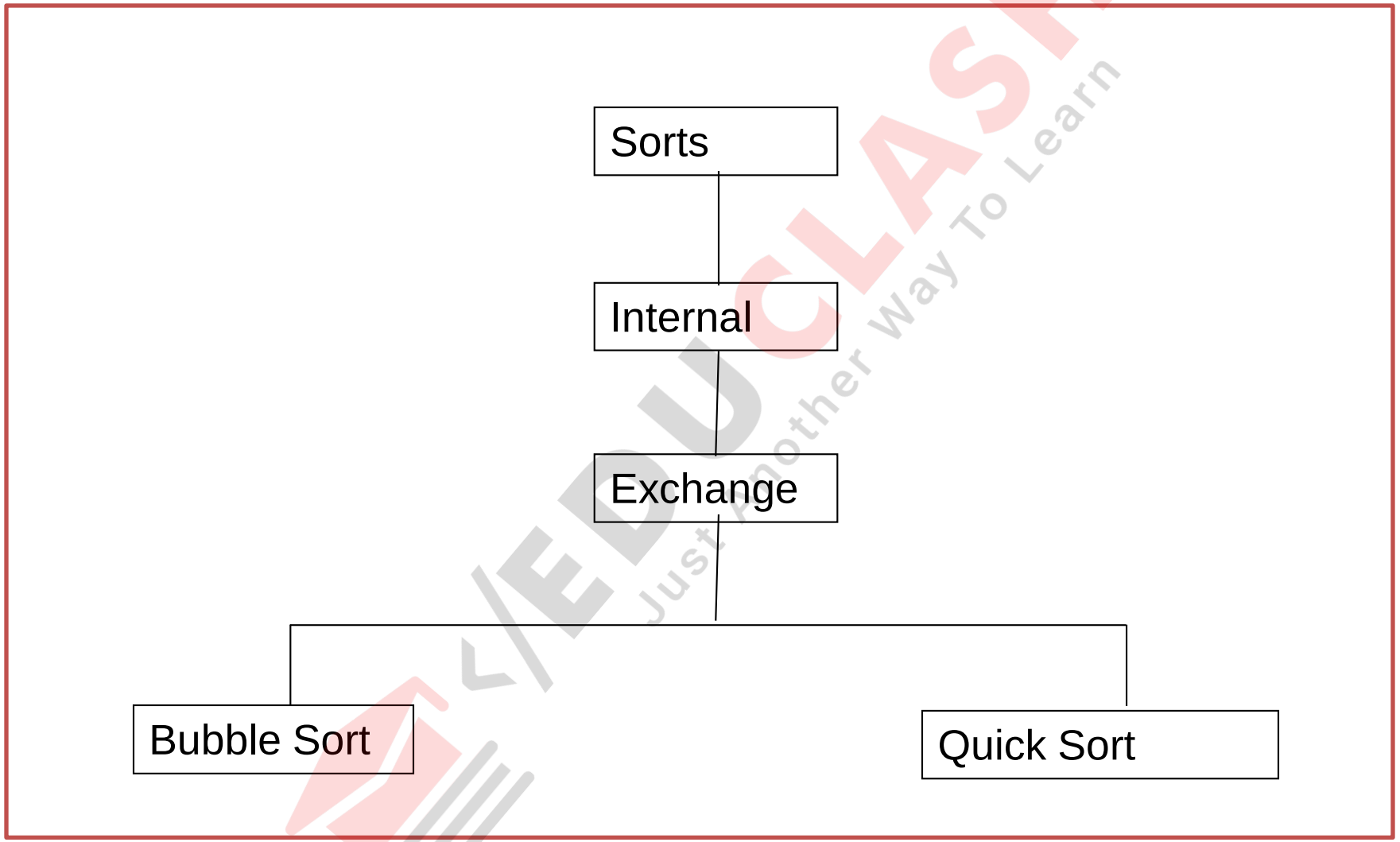
1. build_heap(A)
2. Repeat a,b,c for i=n to 1 step -1

   a. swap(A[0], A[n])

   b. n=n-1

   c. reheapdown(A,0,n)

Exchange Sort



Bubble Sort

- In bubble sort , consecutive items are compared and exchanged on each pass through the list.
- In bubble sort, in each pass through the data, **the smallest element is bubbled to the beginning of the unsorted segment array.**
- **Bubble sort algorithm from the low end and bubbled up i.e. bubbles to the largest element**

void bubble_sort(int *list,int noofelements)

1. [Initialize]

   curdata,nextdata,temp;

2. Repeat for curdata = 0 to curdata<(noofelements -1)

   a. Repeat 1,2 for nextdata= 0 to

      nextdata< (noofelements- 1)- currentdata)

      1. if (list[nextdata]>list[nextdata+1])

         swap the values

      2. increment nextdata

3. Call display_sorted_list(list,noofelements)

Efficiency of Bubble Sort

- The outer loop executes n times(from 0 to noofelements -1).
- The inner loop is dependant on the outer loop(**nextdata<noofelements-currentdata)**,we have a dependant quadratic loop.
  - The efficiency is **f(n)=(n)(n+1)/2**
  - in big-O notation the efficiency of bubble sort is **$O(n^2)$**

Quick Sort

- Quick sort is an exchange sort developed by C.A.R Hoare.
- It is more efficient than the bubble sort because fewer exchanges are required to correctly position an element.
- Working

- Each iteration selects an element known as **pivot** divides the list into 3 groups:
  - A partition of elements whose keys are **less than the pivot's key.**
  - **The pivot element** that is placed in the list
  - A partition of elements whose keys are **greater than the pivot's key.**
  - The **sorting then continues by quick sorting the left followed by quick sorting the right partition.**

- diagram
- **Hoare's original algorithm** selected the **pivot key as the first element in the list**

- **R.C Singleton** improved the sort by **selecting the pivot key as the median value of three elements.**
- Each pass in the quick sort divides the list into 3 parts:
  - A list of elements smaller than the pivot key
  - The pivot key and
  - A list of elements greater than the pivot key
- **Algorithm**

```
Quick_sort( a[], first ,last)
1.[Initialize]
    low=first
    high=last
 pivot=a[(low+high)/2]
2. Repeat A,B,C
    while(low<=high)
    A. Repeat step A
      while(a[low]<pivot)
           A. low=low+1
     B. Repeat step B
      while(a[high]>pivot)
           A. high=high-1
     C. if (low<=high)
           1.swap low and high
             values
          2. low=low+1
           3.high=high-1
4.if(first<high)
   Quick_sort(a, first, high)
5.if(low<last)
   Quick_sort(a, low,last)
6. End
```

Efficiency of Quick Sort

- The first loop(step 2) in conjunction with step A and B looks at each element in the portion of the array being sorted.Therefore they loop through the list **n times.**
- **The list is divided into 2 sublists roughly of the same size using the pivot. Because the list is divided into 2 the number of loops is logarithmic.**
- **Therefore the efficiency is O(nlogn)**

Merge Sort

Algorithm Merge-Sort(Ar, l, r)

if l < r **then**

1.mid = (l+r)/2

2. Merge-Sort(A, left, mid)

3.Merge-Sort(A, mid+1, right)

4.Merge(A, left, mid +1, right)

Binary Tree Sort

- Binary tree sort makes use of Binary Search Tree(BST)
- **Algorithm of Binary tree sort:**

1. To place an element at the appropriate position, the element is compared with node element.

Repeat A while all elements are placed in the tree

A. If the new element <= element in the root node then

it is moved to the left branch

else

it is moved to the right branch

2. To get the tree in the sorted order, the inorder traversal is applied.

- Complexity of Binary Tree Sort:
  - Worst case complexity is $O(n^2)$
  - Best case complexity is $O(n \log_2 n)$
- Example

Radix Sort

- Used to sort a large list of name alphabetically.
- **Algorithm of radix sort**
  1. **In the first iteration, the elements are picked up and kept in various pockets by checking their unit's digit.**
  2. **The data is then collected from pocket 0 to pocket 9 and they are given as input to sorter.**
  3. **In the second iteration, the ten's digit are sorted.**
  4. **Step 2 is repeated**
  5. **In the second iteration, the hundred's digit are sorted.**
  6. **Step 2 is repeated once again**

- **Complexity of Radix sort:**
  - **The time requirement depends on the number of digits and the number of elements in the file. The loop is traversed "m times" for each digit and the inner loop is traversed "n times" for each element in the file.**
  - **The sort is approx O(m*n) .**
  - **m approximates to $\log_2 n$ and so that O(m*n) approximates to O(n $\log_2 n$).**
- **Examples**

Assignment

- Find out efficiency of Merge Sort
- Compare the efficiencies of bubble, selection and insertion sort. Which is the most efficient?