# Binary Search

Binary_search( list, key, n)
where

key → To which we have to search in the list.
n → Represents no. of elements in the list.
list → Represents list of the elements.

Step 1: [Initialize]
low = 1
high = n
flag = 0

Step 2: Repeat through step 4 while (low ≤ high)
Step 3: mid = (low + high) / 2
Step 4: if( key < list[mid])
then
high = mid - 1
else if (element > list [mid])
then
low = mid + 1
else if( key == list[mid])
o/p "search is successful" & location of the element is mid
flag = 1
return
Step 5: if(flag == 0)
o/p "search is unsuccessful", return

Complexity:

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

$\therefore \quad O(\log_2 n)$

# Bubble Sort

Bubble_sort (n, list)
where:
n → Represents the no. of elements in the list
Step 1: [Initialize]
i = 1
Step 2: Repeat through step 7 while (i<n)
Step 3: j = 1
Step 4: Repeat through step 6 while (j<n-i)
Step 5: if ( list[j+1] < list [j])
i) temp = list[j]
ii) list[j] = list[j+1]
iii) list[j+1] = temp
Step 6: j = j+1
Step 7: i = i+1
Step 8: Exit

Complexity OR Efficiency

$$E(n) = \sum_{i=1}^{n} n-i$$

$$= (n-1)+(n-2)+ \ldots + 2+1$$

$$= n(n-1)/2$$

$$= O(n^2) - O(n/2)$$

ie. $O(n^2)$

# Insertion Sort:

Insertion_sort (list, n)
where:
list → Represents the list of elements.
n → Represents the no of elements in the list.
Step 1: [Initialize]
list[0] = 0
Step 2: Repeat through step 3 through 5
for i = 1, 2, 3, 4, ..., n]
i> temp = list[i]
ii> pointer = i-1
Step 3: while (temp < list[pointer])
i> list[pointer+1] = list[pointer]
ii> pointer = pointer - 1
Step 4: list[pointer] = temp
Step 5: Exit.



Efficiency:

$$E(n) = \sum_{i=1}^{n} i-1$$

$$= 1+2+3+4+\dots+n-1$$

$$= n(n-1)/2$$

$$= (n^2-n)/2$$

$$= O(n^2)$$

# Quick Sort (Recursive)

Q-sort(list, first, last)
where:
list → Repesents the list of elements.
first → Represents the position of the first element in the list
last → Represents the position of the last elements in the list.

step 1: [Initially]
low = first
high = last
pivot = first, last or middle elemen
step 2: Repeat through step 7 while (low ≤ high)
step 3: Repeat step 4 while (list[low] < pivot)
step 4: low = low + 1
step 5: Repeat step 6 while (list[high] > pivot)
step 6: high = high - 1
step 7: if (low ≤ high)
i) temp = list [low]
ii) list[low] = list [high]
iii) list[high] = temp
iv) low = low + 1
v) high = high - 1
step 8: if (first < high) Q-sort(list, first, high)
step 9: if (low < last) Q sort(list, low, last)
step 10: exit

## Selection sort:

selection_sort(n, list)

where:

n → Represents size of the list.

list → Represents the list of elements.

step 1: Repeat through step 5 for index = 1, 2, 3, ... n-1

step 2: min = index

step 3: Repeat through step 4 for k = index+1, 2, 3, ... n-1

step 4: if( list[min] > list[k])
min = k

step 5: if(min ≠ index)
i> temp = list[index]
ii> list[index] = list[min]
iii> list[min] = temp.

step 6: Exit



## Efficiency:

$$E(n) = \sum_{index=0}^{n-1} n - index - 1$$

Let $k = n - index - 1$, $k = n-1$ at $index = 0$.
$k = 1$ at $index = n-1$.

$$\therefore E(n) = \sum_{k=0}^{n-1} k$$

$$= n(n-1)/2$$

$$= O(n^2)$$

## Shell sort

shell-sort(list, n)
where:
list → Represents the list of elements.
n → Represents the size of the list.
step1: [initialize]
gap = 5, 3, 1
step2: Repeat through step 6 while (gap = 5, 3, 1)
step3: swap = 0
step4: Repeat through step 6 while (swap)
step5: Repeat through step 6 for i = 0, 1, 2, 3 --
& i < (n - gap)
step6: if( list[i] > list[i + gap])
i) temp = list[i]
ii) list[i] = list[i + gap]
iii) list[i + gap] = temp
iv) swap = 1
step7: O/p list element (sorted)
step8: Exit



## Heap Sort

creat heap(list, n)
where:
list → Represents the list of elements
n → Represents the no. of elements in the list
step1: [Build heap]
Repeat through step 7 for k = 2, 3, 4 -- n
step2: [Initialize]
i) i = k
ii) temp = list[k]
step3: [obtain parent of new elements]
i = i / 2
step4: Repeat through step 6 while (i > 1) &
(temp > list[j])

list[i] = list[j]

Step 6: [obtain next parent]

i) i = j

ii) i = i/2

iii) if (j<1) then j = 1

Step 7: [copy new element value into its proper place]

list[i] = temp

Step 8: Return.

Step 9: Exit.

## Heap Sort Algorithm:

Heap_sort(list, n)

where:

list → Represents the list of elements

n → Represents the no. of elements in the list.

Step 1: [Create initial heap]

call create_heap (list, n)

Step 2: [Start sort]

Repeat through step 10 for k = n, n-1, ---, 2

Step 3: [Exchange elements]

list[1] = list[k]

Step 4: i) temp = list[1]

ii) i = 1

iii) j = 2

Step 5: [find index of largest child of new element]

If (j+1<k) then

If (list[j+1] > list[j])

then j = j+1

~~[Interchange element]~~

~~list[i] = list[j]~~

~~[obtain left child]~~

~~i) i = j~~

~~ii) j = 2*i~~

step6: [Reconstruct the new heap]
Repeat through step10 while (j ≤ k-1) & (List[j] > temp)

step7: [Interchange element]
List[i] = List[j]

step8: [Obtain left child]
i) i = j
ii) j = 2*i

step9: [obtain index of next largest child]
If j+1 < k
if (List[j+1] > List[j]) then
j = j+1
else if (j > n) then
j = 1

step10: [copy element into its proper place]
List[j] = temp

step11: Exit

Radix Sort

Repeat through step 6 for each digit in the key.
Initialize the pockets.
Repeat through step 5 until end of the linked list.
Obtain the next digit of the key.
Insert the element in appropriate pocket.
Combine the pockets to form a new linked list.

Note: pockets are queue[0], queue[1] ... queue[9]

preorder (Root, Left, Right)
VLR

preorder (node)
step 1: [Do through step 3]
If (Node ≠ null)
Step 2 Output Info [Node]
Step 2: Call preorder (Left_child [Node])
step 3: Call preorder (Right_child [Node])
step 4: Exit

Inorder (Left, Root, Right)
LVR

Inorder (node)
Step 1: [Do through step 4]
If (Node ≠ NULL)
step 2: Call Inorder (Left_child [Node])
step 3: output Info [Node]
step 4: Call Inorder [Right_child [Node])
step 5: Exit.

Postorder (Left, Right, Root)
LRV

Postorder (Node)

Step 1: [Do through step 4]
If (Node ≠ NULL)
step 2: Call Postorder (Left_child [Node])
step 3: Call Postorder (Right_child [Node])
step 4: Output Info [Node]
step 5: Exit.