1. What is the additional benefit of a circular queue? Illustrate with at least 2 practical examples (programming/application) where a circular queue is used. Write the enque and deque functions/pseudocode (not the program) for array and linked list-based implementation of a circular queue ?

Solution 1 -
Benefits of Circular Queue-

1. In, Queue sometimes a problem arises that we cannot insert a element even when there is a free space available. This problem is resolved in circular queue.
2. Circular Queue is more efficient in terms of memory utilization because if we delete any element that position can be used later.

Applications of Circular Queue-

1. Memory management: Circular queue is used in memory management.
2. Process Scheduling Algorithms: A CPU uses a queue to schedule processes.
3. Networking Applications - Where server has to serve the multiple client requests in First Come First Serve order .
4. Semaphores- Queues ensures that the process which placed first in the suspended list served first. In short words queue ensures Progress in Process Synchronization.
5. **Linux Kernels-** Queues extensively used in Linux Kernel with different terminology called as Ring-Buffer.

Array Implementation of Circular Queue –

Variables used –

N= Size of Circular Queue

R= Rear Element

F= Front Element

Q [ ] = Array which used to implement the Circular Queue

Code-

1.Enque Function in array implementation-

```
int f= -1; int r= -1;n= 8; q[N]; // Initial values

Circular_enque(q,n,f,r,x) {

if( (r+1) % n == f){
```

```
print (“ Your queue is overflowed”);
exit();
}
if( r== -1){
f++;
r++;
}
else{
r=(r+1)%n;
}
q[r]= x;
}
```

2.Enque Function in array implementation-

```
int f= -1; int r= -1;n= 8; q[N]; // Initial values
Circular_denque (q,n,f,r) {
int y;
if( (f==-1){
print (“ Your queue is underflowed”);
exit();
}
y= q[f];
if( r==f){
f=r=-1;
}
else{
f=(f+1)%n;
}
```

```
return y;
}
```

3.Enque Function in Linked list implementation-

```
void enqueue(int x) //Insert elements in Queue
{
 struct node* n;
 n = (struct node*)malloc(sizeof(struct node));
 n->data = d;
 n->next = NULL;
 if((r==NULL)&&(f==NULL))
{ f = r = n;
 r->next = f;
}
 Else
{
r->next = n;
 r = n;
 n->next = f;
 }
 }
```

4.Deque Function in Linked list implementation-

```
void dequeue() // Delete an element from Queue
{
struct node* t;
 t = f;
 if((f==NULL)&&(r==NULL))
printf("\nQueue is Empty");
else if(f == r)
```

```
{
f = r = NULL;
free(t);
}
Else
{
 f = f->next;
r->next = f;
 free(t);
}
}
```

1. **A sparse matrix is defined as a matrix with most of the elements to be 0. Let us decide the threshold as 70% for the present discussion only, i.e., the matrix with at least 70% zeros. In order to reduce the storage space, usually the non-zero elements of the sparse matrix are stored. Suggest a linked list based representation of a sparse matrix to save the storage space. Can a better scheme be devised?**

Solution 2 – We can represent a matrix in 2 ways –

- Using 2-D Array
- Using Linked List

We use 2-d array representation in case of Dense Graph because in Linked List implementation there is pointer overhead and no random access which makes difficult to traverse.

In case of Sparse graph we usually use Linked list implementation because it takes less space then 2-d array.

Now , let me take a example to propose a scheme to store a sparse matrix –

Matrix size = 5X5

No. of elements = 25

No. of zero's = 18

No. of non zero's = 7

$$A = \begin{matrix} 10 & 0 & 0 & 7 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 9 & 0 & 0 & 0 & 3 \\ 0 & 6 & 0 & 0 & 0 \end{matrix}$$

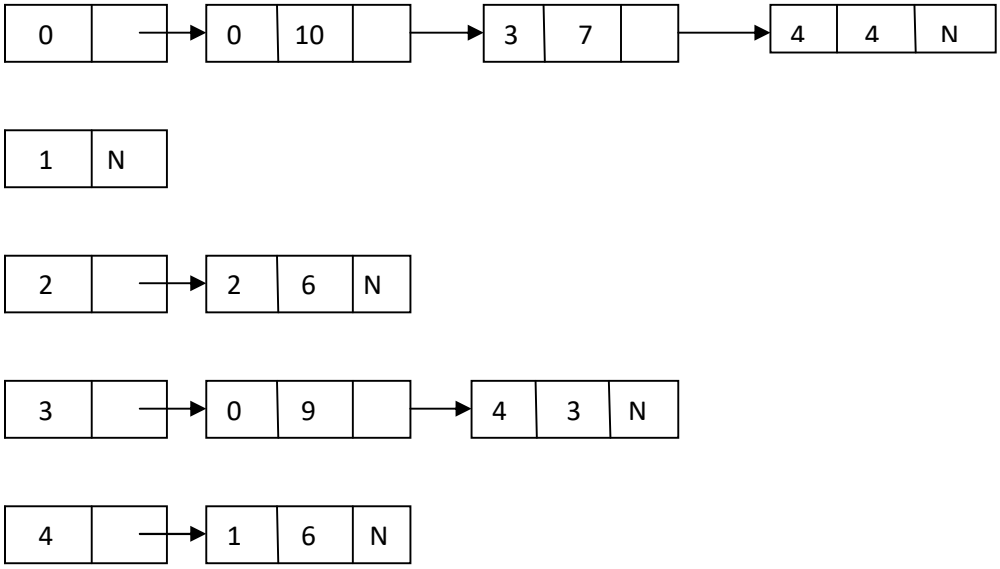
Below I am only storing the non zero elements in terms of Linked List. The starting row_node represents the the row no. and the ptr field contains the memory location of column no. , value which are Ajacent to it.

Node defination-

```
Struct row_node {

int row_no.;

Struct column_node *ptr ;

}
```

```
Struct column_node {
int column_no ;
int data ;
Struct column_node *next ;
}
```

ROW'S



And , I think this is the best way to represent the sparse matrix .

3. How can the bounds be determined for any sorting algorithm? Can a better bound than nlg(n) be obtained?

Solution 3 -

In, comparison based sorting algorithms computation/work done by a CPU is sometimes depends on the input snapshot and sometimes it's same for all the inputs (for example in the case of Merge sort the Best, Avg and Worst case complexities are same for all the input instances).

There is no generalized way finding the Lower and Upper bounds of a algorithm we use different techniques for Iterative and Recursive Algorithms.

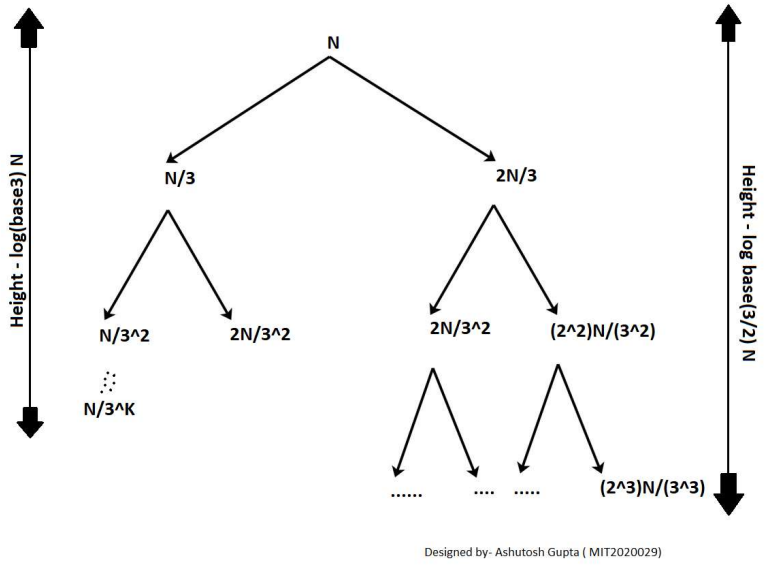
We use Asymptotic Notations to represent the work done by the CPU by using some upper and lower bounds.

1. Big-O Notation (O-Notation):Describes the Avg Case scenario.
2. Omega Notation (Ω-Notation):Describes the Best Case scenario.
3. Theta Notation (Θ-Notation):Describes the Worst Case scenario.

Now I am taking two Hypothetical Examples of Sorting Algorithms to show the technique of finding lower and upper bounds of a sorting algorithm.

1. **Hypothetical Recursive Sorting Algorithm** – A sorting algorithm T which takes a array as input and partition it in two small arrays in O(n) time

    - Recursive Function : T(n) = T(n/3) + T(2n/3) + O(n)
    - Solving this function by Recursive Tree Method :



Time Complexity = (Height of tree x Work done in each level)

T(n) = O(n x log (base 3/2) n)

T(n) = Ω(n x log(base 3) n)

(n x log(base 3) n) <= T(n) <= (n x log (base 3/2) n)

2. **Hypothetical Iterative Sorting Algorithm –** Now, we can take the example of Insertion Sort.

   Working of insertion sort- Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
   Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position. The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result. [wiki]

   - **Best case (when input array is sorted)**
     Comparisons= 1+1+1+1+1+1……..+1+1 {n times} =n
     Swaps=1+1+1+1+1+1……+1+1{n times} =n
     Time complexity for Best case= Ω(n)

   - **Worst case (when input array is reverse sorted)**
     Comparisons= 1+2+3……..+n-1 {n-1 times} =~n^2
     Swaps=1+1+1+1+1+1……+1+1{n-1 times} =~n^2
     Time complexity for Best case= O(n^2)

Solution 3 (Part 2) – This is the very crucial question for computer science community, and the answer is no we cannot sort the array in less than nlog(n).Even God cannot do this in less than nlog(n) because there is a proof which will show below.

Disclaimer- I am not talking about Radix sort and counting sort because these algorithms not give O(n) for any input these algorithms restricted the input in some way.

Proof- Consider a array with n elements and we have n indexes in array and for 0^{th} index place there will be n choices and for 1^{st} index place there will n-1 choices and so on……..

n*n-1*n-2*n-3…….*1 = n! = O(n^n)

If we consider a hypothetical God Algorithm which find the element in logn time in an unsorted array which is not possible .Then also our algorithm takes n*log(n).

.

References –

1. **https://en.wikipedia.org/wiki/Circular_buffer**
2. **https://cs.stanford.edu/people/eroberts/courses/cs106b/**
3. **http://web.mit.edu/16.070/www/lecture/lecture_5_3.pdf**