ALL ALGORITHMS FOR DS UNITS 1 TO 3

1. **Bubble Sort**

```
void bubble(){
    for(pass=1 to n-1){
        for(i=0 to n-pass-1){
            if (arr[i]>arr[i+1]){
                swap(arr[i]>arr[i+1])
            }
        }
    }
    display(arr);
}
```

2. **Selection Sort**

```
void selection(){
    for(swappos=0 to n-2){
        minpos=swappos
        for(i=minpos+1 to n-1){
            if(arr[i]<arr[minpos]){
                minpos=i
            }
        }
        if(minpos!=swappos){
            swap(arr[minpos], arr[swappos])
        }
    }
}
```

3. **Insertion Sort**

```
void insertion(){
    for(i=1 to n-1){
        key=arr[i]
        j=i-1
        while(arr[j]>key && j>=0){
            arr[j+1]=arr[j]
            j--
        }
        arr[j+1]=key;
    }
}
```

4. **Shell Sort**

```
void shell(){
    gap=n/2
    do{
       do{
           swapped=0;
           for(i=0 to i<n-gap){
               if(arr[i]<arr[i+gap]){
                   swap();
                   swapped=1
               }
           }
       } while(swapped=1)
    } while((gap=gap/2)>=1)
}
```

5. **Converting a matrix into compact (triplet) form**

```
void compact{
      A[0][0]=m, A[0][1]=n
      k=1
      for(i=0 to m-1){
          for(j=0 to n-1){
              if(mat[i][j]!=0){
                  A[k][0]=i
                  A[k][1]=j
                  A[k][2]=mat[i][j]
                  k++
              }
          }
      }
      A[0][2]=k-1
}
```

6. **Simple transpose**

```
void s_transpose(){
      m=A[0][0], n=[0][1], t=[0][2]
      B[0][0]=n, B[0][1]=m, B[0][2]=t
      if t<=0 return
      q=1
      for(col=0 to n-1){
          for(p=1 to t){
              if(A[p][1]==col){
                  B[q][0]=A[p][1]
                  B[q][1]=A[p][0]
                  B[q][2]=A[p][2]
                  q++
              }
          }
      }
}
```

7. **Fast transpose**

```
void fast_transpose(){
      m=A[0][0], n=[0][1], t=[0][2]
      B[0][0]=n, B[0][1]=m, B[0][2]=t
      if t<=0 return

      declare arrays S and T
      for(i=0 to n-1){
          S[i]=0
      }

      for(p=1 to t){
          index=A[p][1]
          S[index]++
      }

      T[0]=1

      for(i=1 to n){
          T[i]=T[i-1]+S[i-1]
      }

      for(p=1 to t){
          index=A[p][1]
          pos=T[index]
          B[pos][0]=A[p][1]
          B[pos][1]=A[p][0]
          B[pos][2]=A[p][2]
          T[index]++
      }
}
```

8. **Check if stack is empty**

```
int isEmpty(){
    if (top == -1) return 1;
    return 0;
}
```

9. **Check if stack is full**

```
int isFull(){
      if (top == size-1) return 1;
      return 0;
}
```

10. **Push into a stack**

```
void push(element){
      if (isFull){
             printf(“Stack is full”)
      }
      else{
            top++
            S[top] = element
      }
}
```

11. **Pop from a stack**

```
int pop(S[]){
      if (isEmpty){
             printf(“Stack is empty”)
      }
      else{
            value = S[top]
            top--
            return value
      }
}
```

12. **Infix to postfix**

```
void in_post(){
    int k=0;
    for(i=0 to len-1){
        tkn=inexp[i];

        if(tkn==operand){
            postexp[k]=tkn;
            k++;
        }

        else if(tkn=='('){
            push(tkn);
        }

        else if(tkn==')'){
            tkn = pop();
            while(tkn != '('){
                postexp[k] = tkn;
                k++;
                tkn = pop();
            }
        }
        else{
            while(!isEmpty && isp >= icp){
                postexp[k] = pop();
                k++;
            }
            push(tkn);
        }
    } // end for

    while(!isEmpty){
        postexp[k] = pop();
        k++;
    }
}
```

13. **Infix to prefix**

```
void in_post(){
    int k=0;
    for(i=0 to len-1){
        tkn=inexp[i];

        if(tkn==operand){
            postexp[k]=tkn;
            k++;
        }

        else if(tkn=='('){
            push(tkn);
        }

        else if(tkn==')'){
            tkn = pop();
            while(tkn != '('){
                postexp[k] = tkn;
                k++;
                tkn = pop();
            }
        }
        else{
            while(!isEmpty && isp > icp){
                postexp[k] = pop();
                k++;
            }
            push(tkn);
        }
    } // end for

    while(!isEmpty){
        postexp[k] = pop();
        k++;
    }
}
```

14. **Postfix to infix**

```
void post_infix(){
    for(i=0 to len-1){
        tkn=postexp[i];
        if(tkn==operand){
            push(tkn);
        }
        else{
            op2=pop();
            op1=pop();
            push(strcat('(', op1, tkn, op2, ')'));
        }
    }
    inexp=pop();
}
```

15. **Postfix to prefix**

```
void post_pre(){
    for(i=0 to len-1){
        tkn=postexp[i];
        if(tkn==operand){
            push(tkn);
        }
        else{
            op2=pop();
            op1=pop();
            push(strcat(tkn, op1, op2));
        }
    }
    preexp=pop();
}
```

16. **Prefix to infix**

```
void pre_infix(){
    for(i=len-1 to 0){
        tkn=preexp[i];
        if(tkn==operand){
            push(tkn);
        }
        else{
            op1=pop();
            op2=pop();
            push(strcat('(', op1, tkn, op2, ')'));
        }
    }
    inexp=pop();
}
```

17. **Prefix to postfix**

```
void pre_post(){
    for(i=len-1 to 0){
        tkn=preexp[i];
        if(tkn==operand){
            push(tkn);
        }
        else{
            op1=pop();
            op2=pop();
            push(strcat(op1, op2, tkn));
        }
    }
    postexp=pop();
}
```

18. **Check if a queue is empty**

```
int isEmpty(){
    if(rear==front) return 1;
    return 0;
}
```

19. **Check if a queue is full**

```
int isFull(){
    if(rear=size-1) return 1;
    return 0;
}
```

20. **Enqueue an element in a queue**

```
void enqueue(element){
    if(isFull){
        printf("Queue is full")
    }

    else{
        rear++
        queue[rear]=element;
    }
}
```

21. **Dequeue an element from a queue**

```
int dequeue(){
    if(isEmpty){
        printf("Queue is Empty")
    }
    else{
        front++
        value=queue[front]
        return value
    }
}
```

22. **Check if a circular queue is empty**

```
int isEmpty(){
    if(rear==front){
        return 1
    }
    else{
        return 0
    }
}
```

23. **Check if a circular queue is full**

```
int isFull(){
    if(rear+1)%n==front{
        return 1
    }
    else{
        return 0
    }
}
```

24. **Enqueue an element in a circular queue**

```
void enqueueCQ(element){
    if(isFull){
        printf("Circular Queue is full")
    }
    else{
        rear=(rear+1)%n
        queue[rear]=element
    }
}
```

25. **Dequeue an element from a circular queue**

```
int dequeue(){
    if(isEmpty){
        printf("Circular Queue is Empty")
    }
    else{
        front=(front+1)%n
        value=queue[front]
        return value
    }
}
```