****

Krushkals Algorithm

```
public class Kruskal {

    public void KruskalAlgo(Edge edges[],int vertices) {

        int mst[][] = new int[vertices][vertices];

        Arrays.sort(edges);

        //i
        int edgeCounter = 0;
        int edgeTaken = 0;

        int parent[] = new int[vertices];
        int rank[] = new int[vertices];

        for(int i=0;i<vertices;i++) {
            parent[i]=-1;
            rank[i] = 0;

        }

        while(edgeTaken !=  vertices-1){
            Edge e = edges[edgeCounter];
            if(!isCyclic(e.u,e.v,parent)) {
                union(findParent(e.u, parent), findParent(e.v, parent),
parent, rank);
                mst[e.u][e.v] = e.w;
                edgeTaken++;

            }


            edgeCounter++;
        }

    }
```

****

Topological sorting

```
public class TopologicalSorting {

    ArrayList<Integer> Sorted = new ArrayList<Integer>();

    public boolean isCyclic(int n,ArrayList<ArrayList<Integer>> adj) {

        int indegree[] = new int[n];

        for(int i=0;i<n;i++) {
            for(Integer j : adj.get(i)) {
                indegree[j]++;
            }
        }

        Queue<Integer> q = new LinkedList<Integer>();
        for(int i=0;i<indegree.length;i++) {
            int degree = indegree[i];
            if(degree==0) {
                q.add(degree);
            }
        }

        int count=0;
        while(!q.isEmpty()) {
            int node = q.poll();
            Sorted.add(node);
            count++;

            for(Integer j : adj.get(node)) {
                indegree[j]--;
                if(indegree[j]==0) {
                    q.add(j);
                }

            }


        }
        if(count==n) {
            System.out.println(Sorted);
            return false;
        }else {

            return true;
        }  }  }
```

****

Balanced Paranthesis

```
public boolean balancedParanthesis(String exp) {

        Stack<Character> st = new Stack<Character>();

        for(int i=0;i<exp.length();i++) {
            char c = exp.charAt(i);
            if(c=='{' || c=='(' || c=='[') {
                st.push(c);
            }
            else if(c=='}' || c==')' || c==']' ) {
                if(!st.empty() && (c=='}' && st.peek()=='{') || (c==')' &&
st.peek()=='(') || (c==']' && st.peek()=='[') ) {
                    st.pop();
                }
                else {
                    st.push(c);
                }
            }

        }

        if(st.empty()) {
            return true;
        }
        else {
            return false;
        }
    }
```

****

Alternate Alphabets

```
public class AlphabetsAlternate {

    public void printOdd() {
        try {
            char c = 'a';


            while(c<='z') {
                System.out.println(Thread.currentThread().getName()+" "+c);
                c++;
                c++;

                Thread.sleep(1000);
            }
        }catch (Exception e) {
            System.out.println(e);
        }


    }

    public void printEven() {


                    try {
                        char c = 'b';


                        while(c<='z') {
                            System.out.println(Thread.currentThread().getName(
)+" "+c);

                            c++;
                            c++;

                            Thread.sleep(1500);
                        }

                    } catch (InterruptedException e) {
                        System.out.println(e);
                    }


    }
    public static void main(String[] args) {
        AlphabetsAlternate obj = new AlphabetsAlternate();
Thread t1 = new Thread(new Runnable() {
```

```
            public void run() {
                //print odd
                obj.printOdd();
            }

        },"Thread-1");

        Thread t2 = new Thread(new Runnable() {

            public void run() {
                //print even
                obj.printEven();
            }

        },"Thread-2");


        t1.start();
        t2.start();
    }
}
```

****

Odd Even two threads

```
public class OddEvenThreads {

    public void printOdd(int limit) {
        try {
            for(int i=1;i<limit;i++) {
                if(i%2!=0) {
                    System.out.println(Thread.currentThread().getName()+" "+
i);
                }
                Thread.sleep(1000);
            }
        }catch (Exception e) {
            System.out.println(e);
        }


    }

    public void printEven(int limit) {


                    try {
                        for(int i=1;i<limit;i++) {
                            if(i%2==0) {
                                System.out.println(Thread.currentThread().getN
ame()+" " + i );
                            }
                            Thread.sleep(1000);
                        }
                    } catch (InterruptedException e) {
                        System.out.println(e);
                    }


    }


    public static void main(String[] args) {

        OddEvenThreads obj = new OddEvenThreads();

        Thread t1 = new Thread(new Runnable() {

            public void run() {
                //print odd
                obj.printOdd(15);
```

```
            }

        },"Thread-1");

        Thread t2 = new Thread(new Runnable() {

            public void run() {
                //print even
                obj.printEven(15);
            }

        },"Thread-2");


        t1.start();
        t2.start();
    }
}
```

****

Quick Sort

```
**************************************************************
        Quick Sort
**************************************************************
void quicksort(int a[],int start,int end){
    int p = partition(a,start,end);
    quicksort(a,start,p-1);
    quicksort(a,p+1,end);
}
```

T(n) = Time taken by 1 partition * no of partitions

best case :- when pivots are placed in theri right place in every iteration done

We have time taken for one partion = n

let k be no of partitions to be done

Total time for all partitions can be written as

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

Putting log on both sides

$$\log_2(n) = k \log_2(2)$$

$$k = \log_2(n)$$

Now

T(n) = Time taken by 1 partition * no of partitions

T(n) = n * k

Putting the value of k we get

$$T(n) = n * \log_2(n)$$

$$T(n) = O(n \log(n))$$

worst case :- when array is in ascending order or descending order

We have time taken for one partion = n (where n is size of array)

Number of partitions to be done $= \frac{(n-1)}{2}$

Now

T(n) = Time taken by 1 partition * no of partitions

$$T(n) = n * \frac{(n-1)}{2}$$

$$T(n) = \frac{n^2 - n}{2}$$

$$T(n) = O(n^2)$$

****

Bubble Sort

```
        Bubble sort
**************************************************************


void bubbleSort(int arr[])
{
    int n = arr.length;
        for (int i = 0; i < n-1; i++)

            for (int j = 0; j < n-1-i; j++)

                if (arr[j] > arr[j+1])
                {

                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
```

pass

Time = n

Time = (n-1)/2

Comparision

best case :- when array is already sorted

worst case :- when array is sorted in descending order

Time complexity

T(n) = pass * comparision

T(n) = n * (n-1)/2

$$T(n) = \frac{n^2-n}{2}$$

$T(n) = O(n^2)$

****

Merge Sort

```
**********************************************************************
                    Merge Sort
**********************************************************************

void mergesort(int a[],int start,int end){
    if(start<end){
        int middle = (start+end)/2;
        mergesort(a,start,middle);
        mergesort(a,middle+1,end);
        merge(a,start,middle,end);
    }
}
```

Time = T(n/2)

Time = T(n/2)

Time = n

$T(n) = T(n/2) + T(n/2) + n$

$T(n) = 2T(n/2) + n$ ← 1

substitute n=n/2 in eqn 1

$T(n/2) = 2T(n/4) + n/2$ ← 2

substitute eqn 2 in eqn 1

$T(n) = 2(2T(n/4) + n/2) + n$

$T(n) = 4T(n/4) + 2n$

$T(n) = 2^2 T(n/2^2) + 2n$ ← 3

substitute n=n/4 in eqn 1

$T(n/4) = 2\,T(n/8\ ) + n/4$ ← 4

substitute eqn 4 in eqn 3

$T(n) = 2^2(2T(n/8) + n/4) + 2n$

$T(n) = 2^3 T(n/2^3) + 3n$

$T(n) = 2^i T(n/2^i) + i\,n$ ← 5

we know that $T(1) = 1$

let $n/2^i = 1$ i:e $n = 2^i$

putting log on both sides

$\log_2(n) = i \log_2 2$

$\log_2(n) = i$

eqn 5 becomes

$T(n) = n\,T(1) + \log_2(n)\,n$

$T(n) = n + n \log_2(n)$

$T(n) = O(n \log(n))$

****

Binary Search

```
**********************************************************
        Binary Search
**********************************************************
```

Time = T(n/2)

```
boolean binarySearch(int a[],int start,int end,int data){
    int middle = (start+end)/2;

    if(data<a[middle]){
        binarySearch(a,start,middle,data);
    }
    elseif(data>a[middle]){
        binarySearch(a,middle+1,end,data);
    }
    else if(data==a[middle]){
        return true;
    }
    else{
        return false;
    }

}
```

Best case : when the key to search exactly in the middle
T(n) = O (1)

Worst case : when the key to search exactly in the first index or last index

$T(n) = \log_2(n)$

$T(n) = O(\log(n))$

$T(n) = c + T(n/2)$ ← 1
substitute n=n/2 in eqn 1

$T(n/2) = c + T(n/4)$ ← 2
substitute eqn 2 in eqn 1

$T(n) = 2c + T(n/4)$ ← 3
substitute n=n/4 in eqn 1

$T(n/4) = c + T(n/8)$ ← 4
substitute eqn 4 in eqn 3

$T(n) = 3c + T(n/2^3)$

$T(n) = i\,c + T(n/2^i)$ ← 5
we know that T(1) = 1
let $n/2^i = 1$ i:e n= 2
putting log on both sides

$\log_2(n) = i \log 2$

$\log_2(n) = i$

eqn 5 becomes

$T(n) = \log_2(n)\, c + T(n/2^{\log_2(n)})$

$T(n) = \log_2(n) + T(n)$

****

Binary Search Tree

```
list insertBST(list node,list root) {
        if(root==null) {
            root=node;
        }
        else if(node.data<root.data) {
            root.left = insertBST(node, root.left);
        }
        else if(node.data>root.data){
            root.right = insertBST(node, root.right);
        }
        return root;
    }
```

****

LRU

```
public void put(int key,int value) {
        list node = new list(key,value);

        if(!map.containsKey(node.data)) {
            if(map.size()==cachesize) {
                map.remove(head.data);
                remove(head);
                insert(node);
                map.put(node.data, node);
            }
            else {
                insert(node);
                map.put(node.data, node);
            }
        }
        else {
            remove(node);
            insert(node);
        }
    }
    public int get(int key) {
        list node = map.get(key);

        if(node==null) {
            return -1;
        }
        else{
      return node.value;
        }    }
```

****

Doubly Linked List

```
public void add(int data) {
        list node = new list(data);

        if(head==null) {
            head=node;
            tail=head;
        }
        else {
            tail.right=node;
            node.left=tail;
            tail=node;
        }

    }

    public void delete(int data) {
        list temp =head;

        if(head==null) {
            System.out.println("nothing there");
        }
        else if(temp.data==data) {
            head=head.next;
        }
        else {
            while(temp.right!=null) {
                list prev=temp.left;
                list next = temp.right;
                if(temp.data==data) {
                    prev.right = next;
                    next.left =prev;
                }
            }
        }


    }
```

****

MultiThreading Example

```
class Mutlithreading_by_extending_thread_class extends Thread{
    public void run(){
        try {

             for(int i=1;i<6;i++) {
                   System.out.println(i);
                    Thread.sleep(2000);
                }
        }
        catch (Exception e) {
            System.out.println("Exception is caught");
        }
    }
}
class Multithreading_by_implementing_runnable_interface implements Runnable{
        public void run()
        {
            try {
                 for(int i=55;i<61;i++) {
                       System.out.println(i);
                        Thread.sleep(2000);
                    }

            }
            catch (Exception e) {

                System.out.println("Exception is caught");
            }
        }
    }

public class Multithreading {
    public static void main(String[] args) {



            Thread obj1= new Thread(new
Multithreading_by_implementing_runnable_interface());
            obj1.start();

            Thread obj2 = new Thread(new
Mutlithreading_by_extending_thread_class());
            obj2.start();

}
}
```

****

Circular Queue

```
public void enqueue(int data) {
        Node entry = new Node(data);

        if(head==null) {
            head=entry;
            tail=head;
            tail.next=head;

        }
        else {
            tail.next=entry;
            tail = entry;
            tail.next = head;
        }
    }


    public void dequeue() {

        if(head==null) {
            System.out.println("Nothing present");
            return;
        }
        else {
            head=head.next;
            tail.next=head;
        }

    }
```

****

Single Queue

```
public void enqueue(int data) {
        Node entry = new Node(data);

        if(head==null) {
            head=entry;
            tail=head;
        }
        else {
            tail.next=entry;
            tail = entry;


        }
    }


    public void dequeue() {

        if(head==null) {
            System.out.println("Nothing present");
            return;
        }
        else {
            head=head.next;
        }

    }
```

****

Stack

```
public void push(int data) {
            Node entry = new Node(data);

            if(head==null) {
                head = entry;
            }
            else {
                entry.next = head;
                head= entry;
            }
        }

        public void pop() {

             if(head==null) {
                System.out.println("Stack Underflow");
                return;
             }
             else {
                 head = head.next;
             }

        }

        public int peek() {
            if(head==null) {
                return -1;
            }
            else {
                return head.data;
            }

        }
```

****

Linked List add and remove node

```
public void addNode(int data) {
        list node = new list(data);

        if(head==null) {
            head = node;
             tail = head;
        }
        else {
            tail.next=node;
             tail=node;
        }

    }

    public  void remove(int data) {
        list temp=head;
        list prev=null;

        if(head==null) {
            return ;
        }
        else if(temp.data==data) {
            head = head.next;
        }
        else {
            while(temp.next!=null) {
                prev=temp;
                temp=temp.next;
                if(temp.data==data) {
                    prev.next = temp.next;
                    break;
                }
            }
        }
    }
```

****

Modified Bfs

```
public static void bfs(int graph[][],int source,int destination) {
        int v = graph.length;
        boolean visited[] = new boolean[v];
        Queue<Integer> queue = new LinkedList<Integer>();
        int parent[] = new int[v];

        queue.add(source);
        visited[source]=true;
        while( queue.size()>0) {
            int popped = queue.poll();


                for(int j=0;j<v;j++) {
                    if(graph[popped][j]==1 && !visited[j]) {
                        visited[j]=true;
                        queue.add(j);
                        parent[j]=popped;
                    }
                }


        }


        Queue<Integer> path = new LinkedList<Integer>();
        path.add(destination);
        int prev = destination;
        for(int i=0;i<v;i++) {
             prev = parent[prev];

            if(prev==source) {
                path.add(prev);
                break;
            }
            path.add(prev);
        }

        while(path.size()>0) {
            System.out.println(path.poll());
        }

        }
```

****

Bfs

```
public static void bfs(int graph[][],int source) {
        int v = graph.length;
        boolean visited[] = new boolean[v];
        Queue<Integer> queue = new LinkedList<Integer>();

        queue.add(source);
        visited[source]=true;

        while( queue.size()>0) {
            int popped = queue.poll();
            System.out.println(popped+" ");

                for(int j=0;j<v;j++) {
                    if(graph[popped][j]==1 && !visited[j]) {
                        visited[j]=true;
                        queue.add(j);
                    }
                }


        }
    }
```

****

Dfs

```
    public static void dfs(int graph[][],int source) {
        boolean visited[] =  new boolean[graph.length];
        dfsRecursive(graph,source,visited);

    }

    public static void dfsRecursive(int graph[][],int source,boolean
visited[]) {
         visited[source] = true;
        System.out.println(source + " ");
         for(int i=0;i<graph.length;i++) {
             if(graph[source][i]!=0 && !visited[i]) {
                 dfsRecursive(graph,i,visited);
             }
         }
    }
```

****

Dijkstra ALgorithm

```
public static void dijikstra(int adjMatrix[][]) {
	int v = adjMatrix.length;
	boolean visited[] = new boolean[v];
	int distance[]=new int[v];
	int parent[] = new int[v];
	for(int i=0;i<v;i++) {
		distance[i]=Integer.MAX_VALUE;
	}
	distance[0]=0;

	for(int i=0;i<v;i++) {
		//find vertex with min vertex
		int minVetex = findMinVertex(distance,visited);
		visited[minVetex] = true;

		//explore neighbors
		for(int j=0;j<v;j++) {
			if(adjMatrix[minVetex][j]!=0 && !visited[j]) {
				int newDistance = distance[minVetex]+adjMatrix[minVetex][j];
				if(newDistance<distance[j]) {
					distance[j]=newDistance;
					parent[j]=minVetex;
				}
			}
		}

	}

	for(int i=0;i<v;i++) {
```

```
		System.out.println(i + " "+distance[i]+ " " + parent[i]);
		}
}
public static int findMinVertex(int distance[],boolean visited[]) {
		int minVertex=-1;

		for(int i=0;i<distance.length;i++) {
			if(!visited[i] && (minVertex==-1 || distance[i] < distance[minVertex])) {
				minVertex = i;
			}
		}
return minVertex;
}
```

****

Postfix to Infix

```
public static void  postfix(String expression) {
        Stack<String> stack = new Stack<String>();

        for(int i=0;i<expression.length();i++) {
            char symbol = expression.charAt(i);
            if( (symbol>='a' && symbol<='z') ||  (symbol>='A' && symbol<='Z')
) {
                stack.add(""+symbol);

            }
            else if(symbol=='+' || symbol=='-' || symbol=='*' || symbol=='/')
{
                String op2=stack.pop();
                String op1= stack.pop();
                String newExpr = "(" + op1+symbol+op2 + ")";
                stack.add(newExpr);

            }

        }

        String result = stack.pop();
        System.out.println(result);
    }
```