

1 Binary Search

```
/*Binary Search
** log n
* binary searches for an element in a sorted array
*/
public static boolean BinarySearch(int[] array, int N, int a) {
    int lo = 0;
    int hi = N-1;
    while(lo <= hi) {
        int mid = (int) (((lo + hi) / 2.0) + 0.6);
        if(array[mid] < a) {
            lo = mid+1;
        } else {
            hi = mid-1;
        }
    }
    if(lo < N && array[lo] == a) {
        return true;
    } else {
        return false;
    }
}
```

2 Longest increasing subsequence EASY

```
/*LongestIncreasingSubsequence
*computes the LIS in quadratic time, but is easy to adapted
*/
//This has not been tested yet (adapted from tested C++ Murcia Code)
public static int longestInc(int[] array, int N) {
    int[] m = new int[N];
    for (int i = N - 1; i >= 0; i--) {
        m[i] = 1;
        for (int j = i + 1; j < N; j++) {
            if (array[j] > array[i]) {
                if (m[i] < m[j] + 1) {
                    m[i] = m[j] + 1;
                }
            }
        }
    }
    int longest = 0;
    for (int i = 0; i < N; i++) {
        if (m[i] > longest) {
            longest = m[i];
        }
    }
    return longest;
}
```

3 Longest increasing subsequence FAST

```
/*LongestIncreasingSubsequence
** n*logn
* computes the LongestIncreasingSubsequence using binary search
*/
public static int[] LongestIncreasingSubsequencenlogn(int[] a, int[] p) {
    int[] m = new int[a.length+1];
    int l = 0;
    for(int i = 0; i < a.length; i++) {
        int lo = 1;
        int hi = l;
        while(lo <= hi) {
            int mid = (int) (((lo + hi) / 2.0) + 0.6);
            if(a[m[mid]] < a[i]) {
                lo = mid+1;
            } else {
                hi = mid-1;
            }
        }
        int newL = lo;
        p[i] = m[newL-1];
        m[newL] = i;
        if(newL > l) {
            l = newL;
        }
    }
    int[] s = new int[l];
    int k = m[l];
    for(int i = l-1; i >= 0; i--) {
        s[i] = a[k];
        k = p[k];
    }
    return s;
}
```

4 Next Permutation

```
/*nextPermutation
**n
*returns true if there is another permutation, can also be used to compute the nextPermutation of an array
*/
public static boolean nextPermutation(char[] a) {
    int i = a.length - 1;
    while(i > 0 && a[i-1] >= a[i]) {
        i--;
    }
    if(i <= 0) {
        return false;
    }
    int j = a.length - 1;
    while (a[j] <= a[i-1]) {
        j--;
    }
    char tmp = a[i - 1];
    a[i - 1] = a[j];
    a[j] = tmp;

    j = a.length - 1;
    while(i < j) {
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        i++;
        j--;
    }
    return true;
}
```

5 Held-Karp TSP

```
/* Held Karp
** 2^n*n^2
* Algorithm for TSP, needs 2^n*n space
*/
public static int[] tsp(int[][] graph) {
    int n = graph.length;
    if(n == 1) return new int[]{0};
    //C stores the shortest distance to node of the second dimension
    //first dimension is the bitstring of included nodes on the way
    int[][] C = new int[1<<n][n];
    int[][] p = new int[1<<n][n];
    //initialize
    for(int k = 1; k < n; k++) {
        C[1<<k][k] = graph[0][k];
    }
    for(int s = 2; s < n; s++) {
        for(int S = 1; S < (1<<n); S++) {
            if(Integer.bitCount(S)!=s || (S&1) == 1)
                continue;
            for(int k = 1; k < n; k++) {
                if((S & (1 << k)) == 0)
                    continue;

                //Smk is the set of nodes without k
                int Smk = S ^ (1<<k);

                int min = Integer.MAX_VALUE;
                int minprev = 0;
                for(int m=1; m<n; m++) {
                    if((Smk & (1<<m)) == 0)
                        continue;
                    //distance to m with the nodes in Smk + connection from m to k
                    int tmp = C[Smk][m] +graph[m][k];
                    if(tmp < min) {
                        min = tmp;
                        minprev = m;
                    }
                }
                C[S][k] = min;
                p[S][k] = minprev;
            }
        }
    }

    //find shortest tour length
    int min = Integer.MAX_VALUE;
    int minprev = -1;
    for(int k = 1; k < n; k++) {
        //Set of all nodes except for the first + cost from 0 to k
        int tmp = C[(1<<n) - 2][k] + graph[0][k];
        if(tmp < min) {
            min = tmp;
            minprev = k;
        }
    }

    //Note that the tour has not been tested yet, only the correctness of the min-tour-value
    //backtrack tour
    int[] tour = new int[n+1];
    tour[n] = 0;
    tour[n-1] = minprev;
    int bits = (1<<n)-2;
    for(int k = n-2; k>0; k--) {
        tour[k] = p[bits][tour[k+1]];
        bits = bits ^ (1<<tour[k+1]);
    }
    tour[0] = 0;
    return tour;
}
```

6 Dijkstra

```
/*Dijkstra
 * finds all shortest paths from vertex src
 * does not work with negative weights
 */
public static void dijkstra(Vertex[] vertices, int src) {
    vertices[src].mindistance = 0;
    PriorityQueue<Vertex> queue = new PriorityQueue<Vertex>();
    queue.add(vertices[src]);
    while(!queue.isEmpty()) {
        Vertex u = queue.poll();
        if(u.visited) continue;
        u.visited = true;
        for(Edge e : u.adjacencies) {
            Vertex v = e.target;
            if(v.mindistance > u.mindistance + e.distance) {
                v.mindistance = u.mindistance + e.distance;
                queue.add(v);
            }
        }
    }
}

class Vertex implements Comparable<Vertex> {
    public int id;
    public int mindistance = Integer.MAX_VALUE;
    public LinkedList<Edge> adjacencies = new LinkedList<Edge>();
    public boolean visited = false;

    public int compareTo(Vertex other) {
        return Integer.compare(this.mindistance, other.mindistance);
    }
}

class Edge {
    public Vertex target;
    public int distance;
    public Edge (Vertex target, int distance) {
        this.target = target;
        this.distance = distance;
    }
}
```

7 Fenwick-Tree

```
int[] fwktree = new int[m + n + 1];
// Tree init
for (int i = 1; i < fwktree.length; i++) {
    fwktree = update(i, mn[i], fwktree);
}

public static int read(int index, int[] fenwickTree) {
    int sum = 0;
    while (index > 0) {
        sum += fenwickTree[index];
        index -= (index & -index);
    }
    return sum;
}

public static int[] update(int index, int addValue, int[] fenwickTree) {
    while (index <= fenwickTree.length - 1) {
        fenwickTree[index] += addValue;
        index += (index & -index);
    }
    return fenwickTree;
}
```

8 Edmonds-Karp Fluss

```
public static boolean BFS(int[][] graph, int s, int t, int[] parent) {
    int N = graph.length;
    boolean[] visited = new boolean[N];
    for(int i = 0; i < N; i++) {
        visited[i] = false;
    }
    Queue<Integer> queue = new LinkedList<Integer>();
    queue.add(s);
    visited[s] = true;
    parent[s] = -1;
    while(!queue.isEmpty()) {
        int u = queue.poll();
        if(u == t) return true;
        for(int v = 0; v < N; v++) {
            if(visited[v] == false && graph[u][v] > 0) {
                queue.add(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }
    return (visited[t]);
}

public static int fordFulkerson(int[][] graph, int s, int t) {
    int N = graph.length;
    int[][] rgraph = new int[graph.length][graph.length];
    for(int u = 0; u < graph.length; u++) {
        for(int v = 0; v < graph.length; v++) {
            rgraph[u][v] = graph[u][v];
        }
    }
    int[] parent = new int[N];
    int maxflow = 0;
    while(BFS(rgraph, s, t, parent)) {
        int pathflow = Integer.MAX_VALUE;
        for(int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            pathflow = Math.min(pathflow, rgraph[u][v]);
        }

        for(int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            rgraph[u][v] -= pathflow;
            rgraph[v][u] += pathflow;
        }

        maxflow += pathflow;
    }
    return maxflow;
}
```

9 Floyd-Warshall

```
/*floydWarshall
** n^3
* finds all shortest paths
* paths in array next, distances in ans
*/
public static void floydWarshall(int[][] graph, int[][] next, int[][] ans) {
    for(int i = 0; i < ans.length; i++) {
        for(int j = 0; j < ans.length; j++) {
            ans[i][j] = graph[i][j];
        }
    }
    for (int k = 0; k < ans.length; k++) {
        for (int i = 0; i < ans.length; i++) {
            for (int j = 0; j < ans.length; j++) {
                if (ans[i][k] + ans[k][j] < ans[i][j]
                    && ans[i][k] < Integer.MAX_VALUE && ans[k][j] < Integer.MAX_VALUE) {
                    ans[i][j] = ans[i][k] + ans[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }
}
```

10 Iterative Breitensuche auf AdjMtrx

```
/* BFS_AdjMtrx_Iterativ
** V+E?
* Breitensuche iterativ auf Adjatenzmatrizen
* returns true or false, depending on whether there is a connection between s and g
*/
public static boolean BFSWithoutPathForAdjMatr(int s, int g, int[][] graph) {
    //s being the start and g the goal
    boolean[] visited = new boolean[graph.length];
    for(int i = 0; i < visited.length; i++) visited[i] = false;
    Queue<Integer> queue = new LinkedList<Integer>();
    queue.add(s);
    visited[s] = true;
    while(!queue.isEmpty()) {
        int node = queue.poll();
        if(node == g) return true;
        for(int i = 0; i < graph.length; i++) {
            if(graph[node][i] > 0 && !visited[i]) {
                queue.add(i);
                visited[i] = true;
            }
        }
    }
    return false;
}
```

11 Topologische Sortierung

```
/*
* TopologischeSortierung
* Sortiert einen Grafen (hier als AdjMtrx) topologisch
* Laufzeit: O(V+E)
*/
// l enthaelt alle Knoten topologisch sortiert (Start: 0, Ende= n)
int[] l = new int[n];
int idx = 0;
// s enthaelt alle Knoten, die keine eingehende Kante haben
ArrayList<Integer> s = new ArrayList<Integer>();
// initialisiere s
for (int i = 0; i < n; i++) {
    if (edgesIn[i] == 0) {
```

```

        s.add(i);
    }
}
// Algo Beginn
while (!s.isEmpty()) {
    int node = s.remove(0);
    l[idx++] = node;
    for (int i = 0; i < n; i++) {
        if (adjMtrx[node][i]) {
            adjMtrx[node][i] = false;
            edgesIn[i] -= 1;
            if (edgesIn[i] == 0) {
                s.add(i);
            }
        }
    }
}
}
}

```
