1. $\Theta(1)$
2. $\Theta(\log_2 n)$
3. $\Theta(n)$
4. $\Theta(n \log_2 n)$
5. $\Theta(n^2)$
6. $\Theta(n^2 \log_2 n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$
9. $\Theta(n!)$

Property of asymptotic:

If f(n) = Θ(g(n)), then there exists positive constants c1, c2, n0 such that 0 ⩽ c1.g(n) ⩽ f(n) ⩽ c2.g(n), for all n ⩾ n0
If f(n) = O(g(n)), then there exists positive constants c, n0 such that 0 ⩽ f(n) ⩽ c.g(n), for all n ⩾ n0
If f(n) = Ω(g(n)), then there exists positive constants c, n0 such that 0 ⩽ c.g(n) ⩽ f(n), for all n ⩾ n0
If f(n) = o(g(n)), then there exists positive constants c, n0 such that 0 ⩽ f(n) < c.g(n), for all n ⩾ n0
If f(n) = ω(g(n)), then there exists positive constants c, n0 such that 0 ⩽ c.g(n) < f(n), for all n ⩾ n0

Reflexivity
If f(n) is given then
f(n) = O(f(n))
If f(n) = n3 ⇒ O(n3)

Symmetry:
f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))
If f(n) = n2 and g(n) = n2 then f(n) = Θ(n2) and g(n) = Θ(n2)

Transistivity:
f(n) = O(g(n)) and g(n) = O(h(n)) ⇒ f(n) = O(h(n))
If f(n) = n, g(n) = n2 and h(n) = n3 ⇒ n is O(n2) and n2 is O(n3) then n is O(n3)

Transpose Symmetry:
f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
If f(n) = n and g(n) = n2 then n is O(n2) and n2 is Ω(n)

Since these properties hold for asymptotic notations, analogies can be drawn between functions f(n) and g(n) and two real numbers a and b.
g(n) = O(f(n)) is similar to a ⩽ b
g(n) = Ω(f(n)) is similar to a ⩾ b
g(n) = Θ(f(n)) is similar to a = b
g(n) = o(f(n)) is similar to a < b
g(n) = ω(f(n)) is similar to a > b

Observations:
max(f(n), g(n)) = Θ(f(n) + g(n))
O(f(n)) + O(g(n)) = O(max(f(n), g(n)))

Graph Algorithms

Algorithm	Time Complexity		Space Complexity
	Average	Worst	Worst
Dijkstra's algorithm	O(|E| log |V|)	O(|V|^2)	O(|V| + |E|)
A* search algorithm	O(|E|)	O(b^d)	O(b^d)
Prim's algorithm	O(|E| log |V|)	O(|V|^2)	O(|V| + |E|)
Bellman–Ford algorithm	O(|E| · |V|)	O(|E| · |V|)	O(|V|)
Floyd-Warshall algorithm	O(|V|^3)	O(|V|^3)	O(|V|^2)
Topological sort	O(|V| + |E|)	O(|V| + |E|)	O(|V| + |E|)

Graph Data Structure Operations

Data Structure	Time Complexity					
	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Tree Sort	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Shell Sort	O(n log(n))	O(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Cubesort	O(n)	O(n log(n))	O(n log(n))	O(n)

各种常见算法框架以及时间复杂度

Bubble Sort:

Time Complexity
Best O(n)
Worst O(n 2)
Average O(n 2)
Space Complexity O(1)
Stability Yes

```
def bubblesort(nums):

    for i in range(len(nums)):

        ## FOr the second iteration, the reaason that it use the len(ums) - i - 1
        ##is that for every iteration, the largest
        ## number is always going to be put at the end of the list.
        ##therefore, there is no need to re compare the last two.
        for j in range(len(nums) - i - 1):

            if nums[j] > nums[j + 1]:

                nums[j], nums[j + 1] = nums[j + 1], nums[j]
```

Insertion Sort:

Insertion Sort Complexity	
Best	O(n)
Worst	O(n 2)
Average	O(n 2)
Space Complexity	O(1)
Stability	Yes

```
def insertionSort(nums):

    ## suppose always the first one is sorted. Then do not need to iterate the first
    for i in range(1, len(nums)):

        key = nums[i]
        j = i - 1

        while j >= 0 and key < nums[j]:

            ## move this items to the next position. 往后移位，给前面腾空间.
            nums[j + 1] = nums[j]

            ## 一直移位，直到不用移动了

            j -= 1

        nums[j + 1] = key
```

Merge Sort:

Merge Sort Complexity	
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)
Stability	Yes

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)
#     # alternative
#     L = [0 for i in range(n1)]
#     R = [0 for i in range(n2)]
    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1


def mergeSort_2(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
```

Quick Sort

Quick Sort Complexity	
Best	O(n*log n)
Worst	O(n 2)
Average	O(n*log n)
Space Complexity	O(log n)
Stability	No

```python
def partition(nums, left, right):
    pivot = nums[right]
    pointer = left - 1
    for i in range(left, right):
        if nums[i] <= pivot:
            pointer += 1
            ## remeber, pointer is always to keep the last item that is smaller than pivot, therefore, once nums[i] is smaller
            ## than pivot, pointer plus one then swap nums[i] with nums[pointer]
            nums[pointer] , nums[i] = nums[i], nums[pointer]
    ## this is the place that swap the pivot and the pointer + 1, pointer now is the last one that is smaller than pivot
    ## pointer  + 1 is the first bigger than pivot one, after this one, pivot will be in the right position
    nums[right], nums[pointer + 1] = nums[pointer + 1], nums[right]
    ## after swapping, pointer + 1 is the pivot's current position
    return pointer + 1
```

```python
def quicksort(nums, left, right):
    if left < right:
        ## is to get the pivot's current position, use to divide the origianl nums into two by using partition position.
        parti = partition(nums, left, right)
        quicksort(nums, left, parti- 1)
        quicksort(nums, parti  + 1, right)
```

Heap Sort

Heap Sort Complexity	
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(1)
Stability	No

```python
def heapSort(arr):
    n = len(arr)
     # Build max heap
     # n // 2 is the last root node amongth the all nodes.
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)
    print(arr)

    for i in range(n-1, 0, -1):
        # since the tree has already been a max heap, the first one, arr[0] is always the biggest,
        # Swap it with the last arr[i] , where i start from the last.
        arr[i], arr[0] = arr[0], arr[i]
         # Heapify root element, here, we use i, means we do not include the last items when we heapify, which means remove.
        heapify(arr, i, 0)
```

Binary search:

```java
int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1; // 注意

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
    }
    return -1;
}
```

Graph:

Traverse graph:

```java
// 记录所有路径
List<List<Integer>> res = new LinkedList<>();

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
    // 维护递归过程中经过的路径
    LinkedList<Integer> path = new LinkedList<>();
    traverse(graph, 0, path);
    return res;
}

/* 图的遍历框架 */
void traverse(int[][] graph, int s, LinkedList<Integer> path) {
    // 添加节点 s 到路径
    path.addLast(s);

    int n = graph.length;
    if (s == n - 1) {
        // 到达终点
        res.add(new LinkedList<>(path));
        // 可以在这直接 return，但要 removeLast 正确维护 path
        // path.removeLast();
        // return;
        // 不 return 也可以，因为图中不包含环，不会出现无限递归
    }

    // 递归每个相邻节点
    for (int v : graph[s]) {
        traverse(graph, v, path);
    }

    // 从路径移出节点 s
    path.removeLast();
}
```

```java
// 判断输入的若干条边是否能构造出一棵树结构
boolean validTree(int n, int[][] edges) {
    // 初始化 0...n-1 共 n 个节点
    UF uf = new UF(n);
    // 遍历所有边，将组成边的两个节点进行连接
    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        // 若两个节点已经在同一连通分量中，会产生环
        if (uf.connected(u, v)) {
            return false;
        }
        // 这条边不会产生环，可以是树的一部分
        uf.union(u, v);
    }
    // 要保证最后只形成了一棵树，即只有一个连通分量
    return uf.count() == 1;
}

class UF {
    // 见上文代码实现
}
```

```java
// 记录图是否符合二分图性质
private boolean ok = true;
// 记录图中节点的颜色，false 和 true 代表两种不同颜色
private boolean[] color;
// 记录图中节点是否被访问过
private boolean[] visited;

// 主函数，输入邻接表，判断是否是二分图
public boolean isBipartite(int[][] graph) {
    int n = graph.length;
    color =  new boolean[n];
    visited =  new boolean[n];
    // 因为图不一定是联通的，可能存在多个子图
    // 所以要把每个节点都作为起点进行一次遍历
    // 如果发现任何一个子图不是二分图，整幅图都不算二分图
    for (int v = 0; v < n; v++) {
        if (!visited[v]) {
            traverse(graph, v);
        }
    }
    return ok;
}

// DFS 遍历框架
private void traverse(int[][] graph, int v) {
    // 如果已经确定不是二分图了，就不用浪费时间再递归遍历了
    if (!ok) return;

    visited[v] = true;
    for (int w : graph[v]) {
        if (!visited[w]) {
            // 相邻节点 w 没有被访问过
            // 那么应该给节点 w 涂上和节点 v 不同的颜色
            color[w] = !color[v];
            // 继续遍历 w
            traverse(graph, w);
        } else {
            // 相邻节点 w 已经被访问过
            // 根据 v 和 w 的颜色判断是否是二分图
            if (color[w] == color[v]) {
                // 若相同，则此图不是二分图
                ok = false;
            }
        }
    }
}
```

```java
int minimumCost(int n, int[][] connections) {
    // 城市编号为 1...n, 所以初始化大小为 n + 1
    UF uf = new UF(n + 1);
    // 对所有边按照权重从小到大排序
    Arrays.sort(connections, (a, b) -> (a[2] - b[2]));
    // 记录最小生成树的权重之和
    int mst = 0;
    for (int[] edge : connections) {
        int u = edge[0];
        int v = edge[1];
        int weight = edge[2];
        // 若这条边会产生环，则不能加入 mst
        if (uf.connected(u, v)) {
            continue;
        }
        // 若这条边不会产生环，则属于最小生成树
        mst += weight;
        uf.union(u, v);
    }
    // 保证所有节点都被连通
    // 按理说 uf.count() == 1 说明所有节点被连通
    // 但因为节点 0 没有被使用，所以 0 会额外占用一个连通分量
    return uf.count() == 2 ? mst : -1;
}

class UF {
    // 见上文代码实现
}
```

BFS 框架：

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

DJstra：

```java
class State {
    // 记录 node 节点的深度
    int depth;
    TreeNode node;

    State(TreeNode node, int depth) {
        this.depth = depth;
        this.node = node;
    }
}


// 返回节点 from 到节点 to 之间的边的权重
int weight(int from, int to);

// 输入节点 s 返回 s 的相邻节点
List<Integer> adj(int s);

// 输入一幅图和一个起点 start，计算 start 到其他节点的最短距离
int[] dijkstra(int start, List<Integer>[] graph) {
    // 图中节点的个数
    int V = graph.length;
    // 记录最短路径的权重，你可以理解为 dp table
    // 定义：distTo[i] 的值就是节点 start 到达节点 i 的最短路径权重
    int[] distTo = new int[V];
    // 求最小值，所以 dp table 初始化为正无穷
    Arrays.fill(distTo, Integer.MAX_VALUE);
    // base case，start 到 start 的最短距离就是 0
    distTo[start] = 0;

    // 优先级队列，distFromStart 较小的排在前面
    Queue<State> pq = new PriorityQueue<>((a, b) -> {
        return a.distFromStart - b.distFromStart;
    });

    // 从起点 start 开始进行 BFS
    pq.offer(new State(start, 0));

    while (!pq.isEmpty()) {
        State curState = pq.poll();
        int curNodeID = curState.id;
        int curDistFromStart = curState.distFromStart;

        if (curDistFromStart > distTo[curNodeID]) {
            // 已经有一条更短的路径到达 curNode 节点了
            continue;
        }
        // 将 curNode 的相邻节点装入队列
        for (int nextNodeID : adj(curNodeID)) {
            // 看看从 curNode 达到 nextNode 的距离是否会更短
            int distToNextNode = distTo[curNodeID] + weight(curNodeID, nextNodeID);
            if (distTo[nextNodeID] > distToNextNode) {
                // 更新 dp table
                distTo[nextNodeID] = distToNextNode;
                // 将这个节点以及距离放入队列
                pq.offer(new State(nextNodeID, distToNextNode));
            }
        }
    }
    return distTo;
}
```

```java
int networkDelayTime(int[][] times, int n, int k) {
    // 节点编号是从 1 开始的，所以要一个大小为 n + 1 的邻接表
    List<int[]>[] graph = new LinkedList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new LinkedList<>();
    }
    // 构造图
    for (int[] edge : times) {
        int from = edge[0];
        int to = edge[1];
        int weight = edge[2];
        // from -> List<(to, weight)>
        // 邻接表存储图结构，同时存储权重信息
        graph[from].add(new int[]{to, weight});
    }
    // 启动 dijkstra 算法计算以节点 k 为起点到其他节点的最短路径
    int[] distTo = dijkstra(k, graph);

    // 找到最长的那一条最短路径
    int res = 0;
    for (int i = 1; i < distTo.length; i++) {
        if (distTo[i] == Integer.MAX_VALUE) {
            // 有节点不可达，返回 -1
            return -1;
        }
        res = Math.max(res, distTo[i]);
    }
    return res;
}

// 输入一个起点 start，计算从 start 到其他节点的最短距离
int[] dijkstra(int start, List<int[]>[] graph) {}
```

Network flow:

```
# Ford-Fulkerson algorith in Python

from collections import defaultdict


class Graph:

    def __init__(self, graph):
        self.graph = graph
        self. ROW = len(graph)


    # Using BFS as a searching algorithm
    def searching_algo_BFS(self, s, t, parent):

        visited = [False] * (self.ROW)
        queue = []

        queue.append(s)
        visited[s] = True

        while queue:

            u = queue.pop(0)

            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u

        return True if visited[t] else False

    # Applying fordfulkerson algorithm
    def ford_fulkerson(self, source, sink):
        parent = [-1] * (self.ROW)
        max_flow = 0

        while self.searching_algo_BFS(source, sink, parent):

            path_flow = float("Inf")
            s = sink
            while(s != source):
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Adding the path flows
            max_flow += path_flow

            # Updating the residual values of edges
            v = sink
            while(v != source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow


graph = [[0, 8, 0, 0, 3, 0],
         [0, 0, 9, 0, 0, 0],
         [0, 0, 0, 0, 7, 2],
         [0, 0, 0, 0, 0, 5],
         [0, 0, 7, 4, 0, 0],
         [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0
sink = 5

print("Max Flow: %d " % g.ford_fulkerson(source, sink))
```

Network flow ,