# 7. solution -

**find_bad_version**

```
class Solution:
    def firstBadVersion(self, n):
        left, right = 1, n

        while (left < right):
            mid = left + (right - left) / 2
            if(isBadVersion(mid)):
                right = mid
            else:
                left = mid + 1
        return int(left)
```

**find first and last position**

```
    def searchRange(self, nums: List[int], target: int) -> List[int]:

        def offset(slices, t, desc=False):

            order_weight = 1

            if desc:
                order_weight = -1

            l, r = 0, len(slices) - 1

            if not slices or order_weight * slices[l] > order_weight * t or order_weight * slices[r] <
order_weight * t:
                return 0

            if len(slices) == 1 and slices[0] == t:
                return 1

            while l <= r:

                m = int((l + r) / 2)
                if slices[m] == t and order_weight * slices[m + order_weight] > order_weight * t:
                    return len(slices) - m - 1 if desc else m
                else:
                    l = m + 1

            return 0


        left, right = 0, len(nums) - 1

        while left < right or (left == right and nums[right] == target):
            mid = int((left + right) / 2)
            if nums[mid] > target:
                right = mid
            elif nums[mid] < target:
                left = mid + 1
            else:
                index = [0, 0]
                index[0] = mid - offset(nums[:mid], target, True)
                index[1] = mid + offset(nums[mid+1:], target)
                return index

        return [-1, -1]
```

**Network Delay Time - Accepted 524 ms 16 MB**

```
#        .
# dfs  time limit      ( stack -> heapq() )

    def networkDelayTime(self, times: List[List[int]], N: int, K: int) -> int:

        from collections import defaultdict
        from heapq import heappush, heappop


        def append_sub_nodes(h, sub_nodes):
            sub_nodes = list(filter(lambda x: not visited.__contains__(x[1][1]) or visited[x[1][1]] > visited[x
[1][0]] + x[0], sub_nodes))

            if sub_nodes:
                for sn in sub_nodes:
                    heappush(h, sn)


        # list to graph
        graph = defaultdict(list)

        for source, target, time in times:
            graph[source].append((time, (source, target)))

        visited = {}
        heapq = []
        heappush(heapq, (0, (0, K)))

        while heapq:

            tup = heappop(heapq)
            prev = tup[1][0]
            node = tup[1][1]
            weight = tup[0]

            stored_path_times = visited[prev] if visited.get(prev) else 0
            path_times = stored_path_times + weight

            if node not in visited:
                visited.setdefault(node, path_times)
                append_sub_nodes(h=heapq, sub_nodes=graph[node])

            else:
                if path_times < visited[node]:
                    visited[node] = path_times
                    append_sub_nodes(h=heapq, sub_nodes=graph[node])

        max_sum = max(visited.values())

        if len(visited) != N:
            return -1

        if max_sum <= 0:
            return -1

        return max_sum
```