```
def dfs(root, target):
   stack = [root]
   visited = set()
   while stack:
      cur = stack.pop()
      if cur == target:
         return True
      if cur not in visited:
         visited.add(cur)
         for n in cur.neighbors:
            stack.append(n)

   return False
```

```
def dfs_recursive(cur, target, visited):
   if cur == target:
      return True
   if cur not in visited:
      visited.add(cur)
      for n in cur.neighbors:
         if dfs_recursive(n, target, visited):
            return True
```

```
def backtracking(candidate):
   if find_solution(candidate):
      output(candidate)
      return
   for n in candidate.next:
      if is_valid(n):
         place(n)
         backtracking(n)
         remove(n)
```

```
def binary_search_1(nums, target):
   left = 0
   right = len(nums)

   # when there is an equal sigh in
while statement, it means the search
space is [0, len(nums) - 1]
   while left <= right:
      mid = (left + right) // 2
      if nums[mid] == target:
         return mid
      elif nums[mid] < target:
         left = mid + 1
      else:
         right = mid
   # if only left gets update with + 1,
then mid won't stack.
   # if only right gets update with -1,
then mid will stack, so the while
statement has to be left + 1 < right
```

```
from collections import deque

def bfs(root):
   # keep status or path in queue if needed
   # deque is a doubly linked list
   q = deque([root])  # keep q and child_q when layer
number is needed.
   visited = set()  # visited set is not needed if there is
no cycle or possibility of repeated visits, such as tree.

   while q:
      node = q.popleft()
      if node not in visited:
         visited.add(node)
         for n in node.neighbors:
            q.append(n)
```

```
def binary_search(nums, target):
   if len(nums) == 0:
      return - 1

   left = 0
   right = len(nums) - 1

   while left <= right:
      mid = (left + right) // 2
      if nums[mid] == target:
         return mid
      elif nums[mid] < target:
         left = mid + 1
      else:
         right = mid - 1
   # end condition left > right
   return -1
```

```
def binary_search_2(nums, target):
   if len(nums) == 0:
      return -1

   left, right = 0, len(nums) - 1
   while left + 1 < right:
      mid = (left + right) // 2
      if nums[mid] == target:
         return mid
      elif nums[mid] < target:
         left = mid
      else:
         right = mid

   # Post-processing:
   # End Condition: left + 1 == right
   if nums[left] == target: return left
   if nums[right] == target: return
right
   return -1
```

```
class UnionFind:
  def __init__(self, n):
    self.root = [i for i in range(n)]
    self.rank = [1] * n

  def find(self, i):
    if i == self.root[i]:
      return i
    self.root[i] =
self.find(self.root[i])
    return self.root[i]

  def union(self, i, j):
    pi = self.root[i]
    pj = self.root[j]

    if pi == pj:
      return 0
    if self.rank[pi] > self.rank[pj]:
      self.root[pj] = pi
    elif self.rank[pi] < self.rank[pj]:
      self.par[pi] = pj
    else:
      self.par[pj] = pi
      self.rank[pi] += 1

    return 1

  def connected(self, x, y):
    return self.find(x) == self.find(y)
```

```
  def topologicalSort(graph, V):
    in_degree = [0]*(V)
    for i in graph:
      for j in graph[i]:
        in_degree[j] += 1

    queue = deque()
    for i in range(V):
      if in_degree[i] == 0:
        queue.append(i)

    cnt = 0
    top_order = []
    while queue:

      u = queue.popleft()
      top_order.append(u)

      for i in graph[u]:
        in_degree[i] -= 1
        if in_degree[i] == 0:
          queue.append(i)

      cnt += 1

    if cnt != V: cycle!
```

```
def kruskal_algorithm(n, edges):
  """
  Time Complexity: O(E · logE)
  """
  edges = sorted(edges, key=lambda x: x[-1])
  uf = UnionFind(n)

  ans = []
  for e in edges:
    if uf.union(e[0], e[1]):
      ans.append(e)
      if len(ans) == n - 1:
        return ans
```

```
def prim_algorithm(n, edges):
  """
   Time Complexity: O(E · logE)
   """
  edges_dict = {}
  for n1, n2, w in edges:
    edges_dict[n1] = edges_dict.get(n1, []) + [(n2, w)]
  heap = []
  heapq.heapify(heap)  # use weight in heapq.
  for n2, w in edges_dict[0]:
    heapq.heappush(heap, [w, n2])

  visited = set(0)
  used_edges = 0
  while used_edges < n:
    w, n1 = heapq.heappop(heap)
    if n1 not in visited:
      visited.add(n1)
      for n2, w in edges_dict[n1]:
        heapq.heappush([w, n2])
        used_edges += 1
```

```
import heapq
def dijkstra_algorithm(n, k, graph):
  Time complexity:  O(V + E log(V))
  heap = []
  heapq.heapify(heap)
  heapq.heappush(heap, (0, k))
  visited = set()

  path_len = [float('inf')] * n
  path_len[k] = 0
  while heap:
    current_w, node = heapq.heappop(heap)
    visited.add(node)
    for v, w in graph[node]:
      if v not in visited:
        new_w = current_w + w
        if new_w < path_len[v]:
          path_len[v] = new_w
          heapq.heappush(heap, (new_w, v))
  return path_len
```

```
def merge_sort(nums):
  if len(nums) <= 1:
    return nums

  pivot = int(len(nums) / 2)
  left_list = merge_sort(nums[0:pivot])
  right_list = merge_sort(nums[pivot:])
  return merge(left_list, right_list)


def merge(left_list, right_list):
  l = r = 0
  ret = []
  while l < len(left_list) and r < len(right_list):
    if left_list[l] < right_list[r]:
      ret.append(left_list[l])
      l += 1
    else:
      ret.append(right_list[r])
      r += 1

  # append what is remained in either of the
lists
  ret.extend(left_list[l:])
  ret.extend(right_list[r:])

  return ret
```

```
def quicksort(lst):
  n = len(lst)
  qsort(lst, 0, n - 1)

def qsort(lst, lo, hi):
  if lo < hi:
    p = partition(lst, lo, hi)
    qsort(lst, lo, p - 1)
    qsort(lst, p + 1, hi)

def partition(lst, lo, hi):
  pivot = lst[hi]
  i = lo
  for j in range(lo, hi):
    if lst[j] < pivot:
      lst[i], lst[j] = lst[j], lst[i]
      i += 1
  lst[i], lst[hi] = lst[hi], lst[i]
```