- array
- list
- stack
- queue
    - deque
    - priority queue(heapq)
- hash table
- graph
    - euiler path: 모든 vertex를 한번씩 방문(한붓그리기)
    - hamiltone path: 모든 node를 한번씩 방문(TSP: 외판원 문제)
        - np problem
    - dfs: 깊이 우선 탐색
        - stack연산 활용
    - bfs: 넓이 우선 탐색
        - queue연산 활용
    - dijkstra: 한점 - 모든점간의 최단거리
        - 음수 edge 불가능
        - greedy하게 heapq 활용
            - import collections
            - import heapq
            -
            -
            - n = 4
            - start = 2
            - edges = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
            - graph = collections.defaultdict(list)
            - for s, d, w in edges:
            - graph[s].append([d, w])
            - dist = collections.defaultdict(int)
            - queue = [[0, start]]
            - while len(queue) > 0:
            - weight, vertex = heapq.heappop(queue)
            - if vertex not in dist:
            - dist[vertex] = weight
            - for d, w in graph[vertex]:
            - alt = weight + w
            - heapq.heappush(queue, (alt, d))
            - for d in dist:
            - print(d, dist[d])
    - bellman-ford:
- tree
    - bst
    - preorder, inorder, postorder
- heap
- trie
- sort
    - bubble
        - def bubbleSort(arr):
        - for i in range(len(arr) - 1):

```
        for j in range(len(arr) - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

- merge

```
def mergeSort(arr):
    def merge(left, right):
        result = []
        while len(left) > 0 or len(right) > 0:
            if len(left) > 0 and len(right) > 0:
                if left[0] <= right[0]:
                    result.append(left[0])
                    left = left[1:]
                else:
                    result.append(right[0])
                    right = right[1:]
            elif len(left) > 0:
                result.append(left[0])
                left = left[1:]
            elif len(right) > 0:
                result.append(right[0])
                right = right[1:]
        return result


    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergeSort(arr[:mid])
    right = mergeSort(arr[mid:])
    return merge(left, right)
```

- quick

```
def quickSort(arr, left, right):
    def partition(left, right):
        pivot = right
        small = left
        for compare in range(left, right):
            if arr[compare] < arr[pivot]:
                arr[small], arr[compare] = arr[compare], arr[small]
                small += 1
        arr[small], arr[pivot] = arr[pivot], arr[small]
        return small


    if left < right:
        pivot = partition(left, right)
        quickSort(arr, left, pivot - 1)
        quickSort(arr, pivot + 1, right)
    return arr
```