1 Libraries

1.1 Heap

```
import heapq

a = [1,2,3,4,5,6]
# Heapq defaults to min heap
heapq.heapify(a) # makes a into a min heap object
heapq.heappop(a)

heapq._heapify_max(a) # Makes a into max heap
heapq._heappop_max(a)
```

1.2 Queue

```
import queue

q = queue.Queue()

q.empty()
q.full() # if maxsize specified
q.put(item)
q.get()
q.qsize()

from collections import deque

q = deque()
q.append(1) # push to queue
q[0] # retrieve, but not remove from queue
q.popleft() # remove and return item from queue
```

1.3 Priority Queue

```
import queue
pq = queue.PriorityQueue()

pq.put((10, 'ten'))
pq.put((1, 'one'))
pq.put((5, 'five'))

x = []
while not pq.empty():
    print(pq.get())
```

1.4 Bisect

- `bisect.bisect_left` returns first element that is not less than a targeted value. Returned value will be `len(a)` if not found.
- `bisect.bisect_right` returns first element that is greater than the target value. Returned value will be `len(a)` if not found.

```
# Students sorted by descending GPA, and tie break by name

import bisect

Student = collections.namedtuple("Student", ("name", "gpa"))

def comp_gpa(student):
    return (-student.gpa, student.name)

def search_student(students, target):
    i = bisect.bisect_left([comp_gpa(s) for s in students], comp_gpa(target))
    return 0 <= i < len(students) and students[i] == target
```

1.5 Hash Tables

```
from collection import defaultdict

d = defaultdict(list)
d["k"] # []
```

1.6 Sorting

Implement `__lt__` for python objects to get sorting behaviour.

```
class Student(object):
    def __init__(self, name):
        self.name = name

    def __lt__(self, other):
        return self.name < other.name
```

2 Common Routines

2.1 Arrays

2.1.1 Binary Search

```
def binary_search(A, target):
    low, high = 0, len(A)-1
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

2.1.2 Partition

```
def partition(A, left, right, idx):
    value = A[idx]
    new_pivot_idx = left
    A[idx], A[right] = A[right], A[idx]

    for i in range(left, right):
        if comp(A[i], value):
            A[i], A[new_pivot_idx] = A[new_pivot_idx], A[i]
            new_pivot_idx += 1
        A[right], A[new_pivot_idx] = A[new_pivot_idx], A[right]
    return new_pivot_idx
```

2.2 Linked Lists

2.2.1 Reverse Sub-list

```
def reverse_sublist(L, start, finish):
    dummy_head = sublist_head = ListNode(0, L)
    for _ in range(1, start):
        sublist_head = sublist_head.next

    sublist_iter = sublist_head.next
    for _ in range(finish-start):
        temp = sublist_iter.next
        sublist_head.next, sublist_iter.next, temp.next = \
            temp, temp.next, sublist_head.next

    return dummy_head.next
```

2.2.2 Cycle Finding

```
def has_cycle(head):
    fast = slow = head
    while fast and fast.next and fast.next.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            return True
    return False
```

2.2.3 Reverse Linked List

```
def reverse_linked_list(head):
    prev = None
    curr = head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev
```

2.3 Trees

2.3.1 Traversal

```
def tree_traversal(root):
    if root:
        print("Preorder: %d" % root.data)
        tree_traversal(root.left)
        print("Inorder: %d" % root.data)
        tree_traversal(root.right)
        print("Postorder: %d" % root.data)
```

Iterative:

```
def inorder_traversal(tree):
    s, result = [], []

    while s or tree:
        if tree:
            s.append(tree)
            # Going left.
            tree = tree.left
        else:
            tree = s.pop()
            result.append(tree.data)
            tree = tree.right
    return result
```

Preorder Iterative:

```
def preorder_traversal(tree):
    path, result = [tree], []

    while path:
```

```
        curr = path.pop()
        if curr:
            result.append(curr.data)
            path += [curr.right, curr.left]

    return result
```

2.4 Heap Routines

2.4.1 Top k

```
import heapq

def top_k(k, stream):
    # Entries compared by length
    min_heap = [(len(s), s) for s in itertools.islice(stream, k)]
    heapq.heapify(min_heap)

    for next_string in stream:
        heapq.heappushpop(min_heap, (len(next_string), next_string))
    return [p[1] for p in heapq.nsmallest(k, min_heap)]
```

2.5 Graph Routines

2.5.1 BFS

```
from queue import Queue

def bfs(node):
    q = Queue()
    q.put(node)
    visited = set()
    visited.add(node)

    while not q.empty():
        n = q.get()
        visit(n)

        for neighbour in n.neighbours:
            if neighbour not in visited:
                q.put(neighbour)
                visited.add(neighbour)
```

2.5.2 DFS

```
def dfs(node):
    stack = [node]
    visited = set()
    visited.add(node)
    while stack:
        n = stack.pop()
        visit(n)
        for neighbour in n.neighbours:
            if neighbour not in visited:
            stack.append(neighbour)
            visited.add(neighbour)
```