Contents

1 Misc

1.1 Debug

```
- Pre-Submit:
    - Check output is equivalent to provided test cases
    - Test more cases if not confident
        - Compute small test cases to test against
    - Determine edge-cases
- General:
    - Re-read problem
    - Explain problem & hence solution to teammate (mascot)
    - Have a snack
    - Check type casting in computation and output
    - Division by zero?
- Wrong Answer:
    - Check output format
    - Check boundary test cases
    - Check '==' vs '>=' vs '<='
    - Are values initialized & reset properly?
- Time Limit Exceeded:
    - Calculate time complexity
    - Check loops will always complete
    - Are constant factors reasonable?
    - Is all code necessary?
    - Are there inbuilt functions for the same tasks?
```

1.2 Python

Local variables are accessed faster than global

1.3 Boilerplate

```python
def main():
    pass
if __name__ == "__main__":
    main()
```

1.4 Map

```python
squared = map(lambda x: x**2, [1, 2, 3, 4])
```

1.5 Filter

```python
even = filter(lambda x: x % 2 == 0, [1, 2, 3, 4])
```

1.6 Longest Increasing Subsequence

```python
# Returns size of, LIS, of array of numbers
def longest_increasing_subsequence(n):
    c = [1] * len(n)
    loc = [-1] * len(n)
    for i in range(1, len(n)):
        for j in range(0, i):
            if n[i] > n[j]:
                if c[j] + 1 > c[i]:
                    c[i] = c[j] + 1
                    loc[i] = j
    m = max(c)
    sol = []
    i = c.index(m)
    while loc[i] > -1:
        sol.append(n[i])
        i = loc[i]
    sol.append(n[i])
    return m, sol[::-1]
```

1.7 Longest Common Subsequence

```python
# Finds length of longest common subsequence, order
↪   maintained
def longest_common_subsequence(s1, s2):
    m, n = len(s1), len(s2)
    c = [[0 for j in range(n + 1)] for i in range(m + 1)]
    for i, c_s1 in enumerate(s1):
        for j, c_s2 in enumerate(s2):
            if c_s1 == c_s2:
                c[i + 1][j + 1] = c[i][j] + 1
            else:
                c[i + 1][j + 1] = max(c[i][j + 1], c[i +
↪                   1][j])
    seq = ""
    i, j = m, n
    while i >= 1 and j >= 1:
        if s1[i - 1] == s2[j - 1]:
            seq += s1[i - 1]
            i, j = i - 1, j - 1
        elif c[i - 1][j] > c[i][j - 1]:
            i -= 1
        else:
            j -= 1
    return len(seq), seq[::-1]
```

2 Data Structures

2.1 DefaultDict

```
from collections import defaultdict # no keyerror dict
d = defaultdict(default_value)
```

2.2 Counter

```
from collections import Counter
c = Counter("mississippi") = {'i': 4, 's': 4, 'p': 2, 'm': 1}
c.update("missouri") # adds counts of inp to counts
```

2.3 Double-Ended Queue

```
from collections import deque
q = deque([1, 2, 3, 4, 5, 6])
# q.append(n), q.popLeft()
# q.insert(i, n), q.extend([])
# q.reverse(), q.rotate()
```

2.4 Segment Tree + Example

```
class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.tree = [0] * (4 * self.n)
        self.nums = nums
        self.build(1, 0, self.n - 1)
    def build(self, node, start, end):
        if start == end:
            self.tree[node] = self.nums[start]
        else:
            mid = (start + end) // 2
            left_node = node * 2
            right_node = node * 2 + 1
            self.build(left_node, start, mid)
            self.build(right_node, mid + 1, end)
            self.tree[node] = self.tree[left_node] +
            ↪   self.tree[right_node]
    def query(self, L, R):
        return self._query(1, 0, self.n - 1, L, R)
    def _query(self, node, start, end, L, R):
        if R < start or L > end:
            return 0
        if L <= start and R >= end:
            return self.tree[node]
        mid = (start + end) // 2
        left_sum = self._query(node * 2, start, mid, L, R)
        right_sum = self._query(node * 2 + 1, mid + 1, end,
        ↪   L, R)
        return left_sum + right_sum
    def update(self, index, value):
        self._update(1, 0, self.n - 1, index, value)
    def _update(self, node, start, end, index, value):
        if start == end:
            self.tree[node] = value
            self.nums[index] = value
        else:
            mid = (start + end) // 2
            if start <= index <= mid:
                self._update(node * 2, start, mid, index,
                ↪   value)
            else:
                self._update(node * 2 + 1, mid + 1, end,
                ↪   index, value)
            self.tree[node] = self.tree[node * 2] +
            ↪   self.tree[node * 2 + 1]


data = [3, 7, 2, 8, 5]
tree = SegmentTree(data)
tree.update(2, 4)  # Update: Set the value at index 1 to 2
print(tree.query(1, 4)) # Query total in range [1, 4]
↪   inclusive
```

2.5 Heap

```
# Example implementation of heap
# Practically equivalent to minimum-spanning-tree
# Example problem this solves:
# Given N ropes of different lengths, find the minimum cost
↪   to connect these ropes, cost is the sum of their lengths
import heapq
def min(arr):
    n = len(arr)
    heapq.heapify(arr) # transform arr into heap in-place
    res = 0
    while(len(arr) > 1):
        first = heapq.heappop(arr)
        second = heapq.heappop(arr)
        res += first + second
        heapq.heappush(arr, first + second)
    return res
smallest_k = heapq.nsmallest(k, heap)
largest_k = heapq.nlargest(k, heap)
# Push to heap, pop & return smallest
smallest = heapq.heappushpop(heap, item)
```

3 Maths

3.1 GCD (Euclidean Algorithm)

```
from math import gcd
```

3.2 Extended GCD

```
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = extended_gcd(b % a, a)
        return gcd, y - (b // a) * x, x
```

3.3 Modular Inverse

```
n = pow(a, -1, b) # (n*a)modb == 1
```

3.4 Sieve of Eratosthenes

```
from math import ceil, sqrt
def sieve_of_eratosthenes(n):
    bool_array = [False, False] + [True] * n
    for i in range(2, int(ceil(sqrt(n)))):
        if bool_array[i]:
            for j in range(i * i, n + 1, i):
                bool_array[j] = False
    primes = [i for i in range(n + 1) if bool_array[i]]
    return primes
```

3.5 Miller-Rabin

```
# probability-based primality test- >k more accurate, slower
def miller_rabin(n, k=5):
    if n < 4: return n == 2 or n == 3
    if n % 2 == 0: return False
    d, r = n - 1, 0
    while d % 2 == 0:
        d, r = d // 2, r + 1
    def witness(a):
        x = pow(a, d, n)
        if x == 1 or x == n - 1: return True
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1: return True
        return False
    return all(witness(random.randint(2, min(n - 2, 2 +
    ↪   int(2 * (pow(n.bit_length(), 2)) ** 0.5)))) for _ in
    ↪   range(k))
```

3.6 Pollard's Rho

```
# integer factorization
def pollard_rho(n):
    if n == 1: return 1
    f = lambda x: (x * x + 1) % n
    x, y, d = 2, 2, 1
    while d == 1:
        x = f(x)
        y = f(f(y))
        d = gcd(abs(x - y), n)
    if d == n:
        return pollard_rho(n)
    return d
```

4 Numeric

5 Flow

5.1 Sorting

```
a = [1, 5, 4, 3, 2]
a.sort() # sort a, set a to sorted
sorted(a) # return sorted values, leave a
```

5.2 Ternary Search

```python
# Find max/min of unimodal func on interval [l, r]
def ternary_search(f, l, r, eps=1e-9):
    while abs(r - l) > eps:
        m1 = l + (r - l) / 3
        m2 = r - (r - l) / 3
        # > = min, < = max
        if f(m1) < f(m2):
            l = m1
        else:
            r = m2
    return (l + r) / 2
```

5.3 Binary Search

```python
import bisect
# Returns index of x or -1 if not found
def binary_search(arr, x):
    i = bisect.bisect_left(arr, x)
    if i != len(arr) and arr[i] == x:
        return i
    else:
        return -1
```

5.4 Memoization (+Fibonacci)

```python
# Top-Down DP
# Remember previous computation result, prevent unnecessary
↪   computation
import functools
@functools.lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

5.5 Tabulation

```python
# Bottom-up DP
# Fill a table, then compute solution from table
def fibonacci(n):
    m = [0, 1]
    for i in range(2, n+1):
        m.append(m[i-2] + m[i-1])
    return m[n]
```

6 Graph

6.1 Modelling

Thanks to github.com/prakhar1989/Algorithms

Notes Source: waynedisonitau123

- Maximum/Minimum flow with lower bound / Circulation problem
  1. Construct super source S and sink T.
  2. For each edge (x, y, l, u), connect $x \to y$ with capacity $u-l$.
  3. For each vertex v, denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
  4. If $in(v) > 0$, connect $S \to v$ with capacity $in(v)$, otherwise, connect $v \to T$ with capacity $-in(v)$.
     - To maximize, connect $t \to s$ with capacity ∞ (skip this in circulation problem), and let f be the maximum flow from S to T. If $f \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, the maximum flow from s to t is the answer.
     - To minimize, let f be the maximum flow from S to T. Connect $t \to s$ with capacity ∞ and let the flow from S to T be f'. If $f + f' \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, f' is the answer.
  5. The solution of each edge e is $l_e + f_e$, where f_e corresponds to the flow of edge e on the graph.
- Construct minimum vertex cover from maximum matching M on bipartite graph (X, Y)
  1. Redirect every edge: $y \to x$ if $(x, y) \in M$, $x \to y$ otherwise.
  2. DFS from unmatched vertices in X.
  3. $x \in X$ is chosen iff x is unvisited.
  4. $y \in Y$ is chosen iff y is visited.
- Minimum cost cyclic flow
  1. Consruct super source S and sink T
  2. For each edge (x, y, c), connect $x \to y$ with $(cost, cap) = (c, 1)$ if $c > 0$, otherwise connect $y \to x$ with $(cost, cap) = (-c, 1)$
  3. For each edge with $c < 0$, sum these cost as K, then increase $d(y)$ by 1, decrease $d(x)$ by 1
  4. For each vertex v with $d(v) > 0$, connect $S \to v$ with $(cost, cap) = (0, d(v))$
  5. For each vertex v with $d(v) < 0$, connect $v \to T$ with $(cost, cap) = (0, -d(v))$
  6. Flow from S to T, the answer is the cost of the flow $C + K$
- Maximum density induced subgraph
  1. Binary search on answer, suppose we're checking answer T
  2. Construct a max flow model, let K be the sum of all weights
  3. Connect source $s \to v$, $v \in G$ with capacity K
  4. For each edge (u, v, w) in G, connect $u \to v$ and $v \to u$ with capacity w
  5. For $v \in G$, connect it with sink $v \to t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
  6. T is a valid answer if the maximum flow $f < K|V|$
- Minimum weight edge cover
  1. For each $v \in V$ create a copy v', and connect $u' \to v'$ with weight $w(u, v)$.
  2. Connect $v \to v'$ with weight $2\mu(v)$, where $\mu(v)$ is the cost of the cheapest edge incident to v.
  3. Find the minimum weight perfect matching on G'.
- Project selection problem
  1. If $p_v > 0$, create edge (s, v) with capacity p_v; otherwise, create edge (v, t) with capacity $-p_v$.
  2. Create edge (u, v) with capacity w with w being the cost of choosing u without choosing v.
  3. The mincut is equivalent to the maximum profit of a subset of projects.
- 0/1 quadratic programming

$$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x \bar{y} + \sum_{xyx'y'} c_{xyx'y'} (x\bar{y} + x'\bar{y'})$$

  can be minimized by the mincut of the following graph:
  1. Create edge (x, t) with capacity c_x and create edge (s, y) with capacity c_y.
  2. Create edge (x, y) with capacity c_{xy}.
  3. Create edge (x, y) and edge (x', y') with capacity $c_{xyx'y'}$.

6.2 Undirected Graph

```python
class graph(object):
    DEFAULT_WEIGHT = 1
    DIRECTED = False
    def __init__(self):
        self.node_neighbors = {}
    def __str__(self):
        return "Undirected Graph \nNodes: %s \nEdges: %s" %
        ↪  (self.nodes(), self.edges())
    def add_nodes(self, nodes):
        """Takes a list of nodes as input and adds these to
        ↪  a graph"""
        for node in nodes:
            self.add_node(node)
    def add_node(self, node):
        """Adds a node to the graph"""
        if node not in self.node_neighbors:
            self.node_neighbors[node] = {}
        else:
            raise Exception("Node %s is already in graph" %
            ↪  node)
    def has_node(self, node):
        """Returns boolean to indicate whether a node exists
        ↪  in the graph"""
        return node in self.node_neighbors
    def add_edge(self, edge, wt=DEFAULT_WEIGHT, label=""):
        """Add an edge to the graph connecting two nodes.
        An edge, here, is a pair of node like C(m, n) or a
        ↪  tuple"""
        u, v = edge
        if (v not in self.node_neighbors[u] and u not in
        ↪  self.node_neighbors[v]):
            self.node_neighbors[u][v] = wt
            if (u!=v):
                self.node_neighbors[v][u] = wt
```

```
        else:
            raise Exception("Edge (%s, %s) already added in
            ↪  the graph" % (u, v))
    def add_edges(self, edges):
        """ Adds multiple edges in one go. Edges, here, is a
        ↪  list of tuples"""
        for edge in edges:
            self.add_edge(edge)
    def nodes(self):
        """Returns a list of nodes in the graph"""
        return self.node_neighbors.keys()
    def has_edge(self, node):
        """Returns a boolean to indicate whether an edge
        ↪  exists in the
        graph. An edge, here, is a pair of node like C(m, n)
        ↪  or a tuple"""
        u, v = edge
        return v in self.node_neighbors.get(u, [])
    def neighbors(self, node):
        """Returns a list of neighbors for a node"""
        if not self.has_node(node):
            raise "Node %s not in graph" % node
        return self.node_neighbors[node].keys()
    def del_node(self, node):
        """Deletes a node from a graph"""
        for each in list(self.neighbors(node)):
            if (each != node):
                self.del_edge((each, node))
        del(self.node_neighbors[node])
    def del_edge(self, edge):
        """Deletes an edge from a graph. An edge, here, is a
        ↪  pair like
        C(m,n) or a tuple"""
        u, v = edge
        if not self.has_edge(edge):
            raise Exception("Edge (%s, %s) not an existing
            ↪  edge" % (u, v))
        del self.node_neighbors[u][v]
        if (u!=v):
            del self.node_neighbors[v][u]
    def node_order(self, node):
        """Return the order or degree of a node"""
        return len(self.neighbors(node))
    def edges(self):
        """Returns a list of edges in the graph"""
        edge_list = []
        for node in self.nodes():
            edges = [(node, each) for each in
            ↪  self.neighbors(node)]
            edge_list.extend(edges)
        return edge_list
    def set_edge_weight(self, edge, wt):
        """Set the weight of an edge """
        u, v = edge
        if not self.has_edge(edge):
            raise Exception("Edge (%s, %s) not an existing
            ↪  edge" % (u, v))
        self.node_neighbors[u][v] = wt
        if u != v:
            self.node_neighbors[v][u] = wt
    def get_edge_weight(self, edge):
        """Returns the weight of an edge """
        u, v = edge
        if not self.has_edge((u, v)):
            raise Exception("%s not an existing edge" % edge)
        return self.node_neighbors[u].get(v,
        ↪  self.DEFAULT_WEIGHT)
    def get_edge_weights(self):
        """ Returns a list of all edges with their weights
        ↪  """
        edge_list = []
        unique_list = {}
        for u in self.nodes():
            for v in self.neighbors(u):
                if u not in unique_list.get(v, set()):
                    edge_list.append((self.node_neighbors[u]
                    ↪  [v], (u,
                    ↪  v)))
                    unique_list.setdefault(u, set()).add(v)
        return edge_list
```

6.3 Directed Graph

```
from graph import graph
from copy import deepcopy
class digraph(graph):
    """ Directed Graph class - made of nodes and edges
    inherits graph methods"""
    DEFAULT_WEIGHT = 1
    DIRECTED = True
    def __init__(self):
        self.node_neighbors = {}
    def __str__(self):
        return "Directed Graph \nNodes: %s \nEdges: %s" %
        ↪  (self.nodes(), self.edges())
    def add_edge(self, edge, wt=DEFAULT_WEIGHT, label=""):
        """Add an edge to the graph connecting two nodes.
        An edge, here, is a pair of node like C(m, n) or a
        ↪  tuple
        with m as head and n as tail :  m -> n"""
        u, v = edge
        if (v not in self.node_neighbors[u]):
            self.node_neighbors[u][v] = wt
        else:
            raise Exception("Edge (%s, %s) already added in
            ↪  the graph" % (u, v))
    def del_edge(self, edge):
        """Deletes an edge from a graph. An edge, here,
         is a pair like C(m,n) or a tuple"""
        u, v = edge
        if not self.has_edge(edge):
            raise Exception("Edge (%s, %s) not an existing
            ↪  edge" % (u, v))
        del self.node_neighbors[u][v]
    def del_node(self, node):
        """Deletes a node from a graph"""
        for each in list(self.neighbors(node)):
            if (each != node):
                self.del_edge((node, each))
        for n in self.nodes():
            if self.has_edge((n, node)):
                self.del_edge((n, node))
        del(self.node_neighbors[node])
    def get_transpose(self):
        """ Returns the transpose of the graph
        with edges reversed and nodes same """
        digr = deepcopy(self)
        for (u, v) in self.edges():
            digr.del_edge((u, v))
            digr.add_edge((v, u))
        return digr
```

6.4 Depth-first Search

```
def DFS(gr, s):
    """ Depth first search wrapper """
    path = set([])
    depth_first_search(gr, s, path)
    return path
def depth_first_search(gr, s, path):
    """ Depth first search
    Returns a list of nodes "findable" from s """
    if s in path: return False
    path.add(s)
    for each in gr.neighbors(s):
        if each not in path:
            depth_first_search(gr, each, path)
```

6.5 Breadth-first Search

```
def BFS(gr, s):
    """ Breadth first search
    Returns list of nodes that are "findable" from s """
    if not gr.has_node(s):
        raise Exception("Node %s not in graph" % s)
    nodes_explored = {s}
    q = deque([s])
    while len(q)!=0:
        node = q.popleft()
        for each in gr.neighbors(node):
            if each not in nodes_explored:
                nodes_explored.add(each)
                q.append(each)
    return nodes_explored
```

6.6 Shortest Hops

```
def shortest_hops(gr, s):
    """ Finds shortest number of hops to reach a node from s.
     Returns a dict mapping: destination node from s -> no.
     ↪  of hops
    """
    if not gr.has_node(s):
```

```
        raise Exception("Node %s is not in graph" % s)
    else:
        dist = {}
        q = deque([s])
        nodes_explored = set([s])
        for n in gr.nodes():
            if n == s: dist[n] = 0
            else: dist[n] = float('inf')
        while len(q) != 0:
            node = q.popleft()
            for each in gr.neighbors(node):
                if each not in nodes_explored:
                    nodes_explored.add(each)
                    q.append(each)
                    dist[each] = dist[node] + 1
        return dist
```

6.7 Shortest Path - Dijkstra's

```
def shortest_path(digr, s):
    """ Finds the shortest path from s to every other vertex
    ↪  findable
    from s using Dijkstra's algorithm in O(mlogn) time"""
    nodes_explored = set([s])
    nodes_unexplored = DFS(digr, s) # all accessible nodes
    ↪  from s
    nodes_unexplored.remove(s)
    dist = {s:0}
    node_heap = []
    for n in nodes_unexplored:
        min = compute_min_dist(digr, n, nodes_explored, dist)
        heapq.heappush(node_heap, (min, n))
    while len(node_heap) > 0:
        min_dist, nearest_node = heapq.heappop(node_heap)
        dist[nearest_node] = min_dist
        nodes_explored.add(nearest_node)
        nodes_unexplored.remove(nearest_node)
        for v in digr.neighbors(nearest_node):
            if v in nodes_unexplored:
                for i in range(len(node_heap)):
                    if node_heap[i][1] == v:
                        node_heap[i] =
                        ↪  (compute_min_dist(digr, v,
                        ↪  nodes_explored, dist), v)
                        heapq.heapify(node_heap)
    return dist
def compute_min_dist(digr, n, nodes_explored, dist):
    """ Computes the min dist of node n from a set of
    nodes explored in digr, using dist dict. Used in
    ↪  shortest path """
    min = float('inf')
    for v in nodes_explored:
        if digr.has_edge((v, n)):
            d = dist[v] + digr.get_edge_weight((v, n))
            if d < min: min = d
    return min
```

6.8 Prim's Min-Cost Spanning Tree

```
def minimum_spanning_tree(gr):
    """ Uses prim's algorithm to return the minimum
    cost spanning tree in a undirected connected graph.
    Works only with undirected and connected graphs """
    s = gr.nodes()[0]
    nodes_explored = set([s])
    nodes_unexplored = gr.nodes()
    nodes_unexplored.remove(s)
    min_cost, node_heap = 0, []
    for n in nodes_unexplored:
        min = compute_key(gr, n, nodes_explored)
        heapq.heappush(node_heap, (min, n))
    while len(nodes_unexplored) > 0:
        node_cost, min_node = heapq.heappop(node_heap)
        min_cost += node_cost
        nodes_explored.add(min_node)
        nodes_unexplored.remove(min_node)
        for v in gr.neighbors(min_node):
            if v in nodes_unexplored:
                for i in range(len(node_heap)):
                    if node_heap[i][1] == v:
                        node_heap[i] = (compute_key(gr, v,
                        ↪  nodes_explored), v)
                        heapq.heapify(node_heap)
    return min_cost
def compute_key(gr, n, nodes_explored):
    """ computes minimum key for node n from a set of
    ↪  nodes_explored
    in graph gr. Used in Prim's implementation """
    min = float('inf')
    for v in gr.neighbors(n):
        if v in nodes_explored:
            w = gr.get_edge_weight((n, v))
            if w < min: min = w
    return min
```

7 Geometry

8 Strings