**Python Notes**

1. i = bisect.bisect\_left( list, num ) The returned insertion point i partitions the array a into two halves so that all(val < x for val in a[lo : i]) for the left side and all(val >= x for val in a[i : hi]) for the right side. => Basically it finds the first rightful position in the array .
2. bisect\_right: The returned insertion point i partitions the array a into two halves so that all(val <= x for val in a[lo : i]) for the left side and all(val > x for val in a[i : hi]) for the right side. Basically it finds the last rightful position of the element in the array in case of a tie.
3. Python generators

|  |
| --- |
| def flatten\_list(nested\_list):  for nested\_integer in nested\_list:  if nested\_integer.isInteger():  yield nested\_integer.getInteger()  else:  for integer in flatten\_list(nested\_integer.getList()):  yield integer  def flatten\_list(nested\_list):  for nested\_integer in nested\_list:  if nested\_integer.isInteger():  yield nested\_integer.getInteger()  else:  yield from flatten\_list(nested\_integer.getList())  class NestedIterator:  def \_\_init\_\_(self, nestedList: [NestedInteger]):  # Get a generator object from the generator function, passing in  # nestedList as the parameter.  self.\_generator = self.\_int\_generator(nestedList)  # All values are placed here before being returned.  self.\_peeked = None  # This is the generator function. It can be used to create generator  # objects.  def \_int\_generator(self, nested\_list) -> "Generator[int]":  # This code is the same as Approach 1. It's a recursive DFS.  for nested in nested\_list:  if nested.isInteger():  yield nested.getInteger()  else:  # We always use "yield from" on recursive generator calls.  yield from self.\_int\_generator(nested.getList())  # Will automatically raise a StopIteration.    def next(self) -> int:  # Check there are integers left, and if so, then this will  # also put one into self.\_peeked.  if not self.hasNext(): return None  # Return the value of self.\_peeked, also clearing it.  next\_integer, self.\_peeked = self.\_peeked, None  return next\_integer    def hasNext(self) -> bool:  if self.\_peeked is not None: return True  try: # Get another integer out of the generator.  self.\_peeked = **next(self.\_generator)**  return True  **except: # The generator is finished so raised StopIteration.**  return False |

1. OrderedDict, move\_to\_end(key), popitem(last=False)
2. Custom Exceptions

|  |
| --- |
| class ValidationError(Exception):  def \_\_init\_\_(self, message, errors):  # Call the base class constructor with the parameters it needs  super().\_\_init\_\_(message)    # Now for your custom code...  self.errors = errors  You could also override the \_\_str\_\_ method to print the appropriate error string |

1. cmp\_to\_key in sorted with the key argument. from functools import cmp\_to\_key
2. Prim’s Algorithm – MST

|  |
| --- |
| import heapq  from decimal import Decimal  import collections  class Solution:  def minimumCost(self, n: int, connections: List[List[int]]) -> int:    cost = collections.defaultdict(list)  print( cost )  for city1, city2, price in connections:  cost[city1].append((price, city2))  cost[city2].append((price, city1))    heap = [(0,n)]  visited = set()    total = 0  while heap and len( visited ) < n:  currentCost, currentCity = heapq.heappop( heap )  if currentCity not in visited:  total += currentCost  visited.add( currentCity )  for nextCost, nextCity in cost[ currentCity ]:  if nextCity not in visited:  heapq.heappush( heap, ( nextCost, nextCity ) )    return total if len(visited) == n else -1 |

Correctness – Intuition – If there is any other spanning tree, it can be transformed to the one obtained by the Prim’s algorithm. In the description below, Y1 is the tree which is MST and Y is the result of Prim’s. Y1 can be transformed to Y2 which is also MST and moves one step closer to Y.

A picture containing text, screenshot, font, document

Description automatically generated

1. Kruskal Algorithm – DSU for MST. Sort by edges and run DSU code.

|  |
| --- |
| class UnionFind {  public:  UnionFind(vector<vector<char>>& grid) {  count = 0;  int m = grid.size();  int n = grid[0].size();  for (int i = 0; i < m; ++i) {  for (int j = 0; j < n; ++j) {  if (grid[i][j] == '1') {  parent.push\_back(i \* n + j);  ++count;  }  else parent.push\_back(-1);  rank.push\_back(0);  }  }  }  int find(int i) { // path compression  if (parent[i] != i) parent[i] = find(parent[i]);  return parent[i];  }  void Union(int x, int y) { // union with rank  int rootx = find(x);  int rooty = find(y);  if (rootx != rooty) {  if (rank[rootx] > rank[rooty]) parent[rooty] = rootx;  else if (rank[rootx] < rank[rooty]) parent[rootx] = rooty;  else {  parent[rooty] = rootx; rank[rootx] += 1;  }  --count;  }  }  int getCount() const {  return count;  }  private:  vector<int> parent;  vector<int> rank;  int count; // # of connected components  };  class Solution {  public:  int numIslands(vector<vector<char>>& grid) {  int nr = grid.size();  if (!nr) return 0;  int nc = grid[0].size();  UnionFind uf (grid);  int num\_islands = 0;  for (int r = 0; r < nr; ++r) {  for (int c = 0; c < nc; ++c) {  if (grid[r][c] == '1') {  grid[r][c] = '0';  if (r - 1 >= 0 && grid[r-1][c] == '1') uf.Union(r \* nc + c, (r-1) \* nc + c);  if (r + 1 < nr && grid[r+1][c] == '1') uf.Union(r \* nc + c, (r+1) \* nc + c);  if (c - 1 >= 0 && grid[r][c-1] == '1') uf.Union(r \* nc + c, r \* nc + c - 1);  if (c + 1 < nc && grid[r][c+1] == '1') uf.Union(r \* nc + c, r \* nc + c + 1);  }  }  }  return uf.getCount();  } |

Complexity for M operations on a forest of N vertices is O( Mlog\*N) where log\*N is the inverse Ackerman function – constant – Union by rank with path compression. Idea is that when you unionise, you put the component with less height as the child of bigger height. So with find this will typically be O(logN). If you combine that with path compression it becomes O(log\*N)

The MST complexity will generally be around O(E \* logV) .