1. Implement a trie data structure to efficiently support autocomplete suggestions

```
class TrieNode:
 def init(self):
self.children, self.is_end = {}, False
class Trie:
 def init(self):
self.root = TrieNode()
def insert(self, word):
   node = self.root
   for c in word:
      node = node.children.setdefault(c, TrieNode())
   node.is_end = True
def search(self, prefix):
   node = self.root
   for c in prefix:
      if c not in node.children:
         return []
      node = node.children[c]
   return self._collect(node, prefix)
def _collect(self, node, prefix):
   res = [prefix] if node.is_end else []
   for c, n in node.children.items():
      res += self._collect(n, prefix + c)
   return res
def main():
 trie = Trie()
 while True:
 choice = input("1.Insert 2.Search 3.Exit: ")
 if choice == '1':
 trie.insert(input("Word: ").strip())
 elif choice == '2':
 p = input("Prefix: ").strip()
 print("Suggestions:", trie.search(p) or "None")
 elif choice == '3':
 break
if name == "main":
 main()
```

4.
Implement a chess game application using backtracking.

```
class ChessGame:
 def init(self):
 self.board = [
 list("rnbqkbnr"),
 list("pppppppp"),
 [' '] * 8,
 [' '] * 8,
 [' '] * 8,
 [' '] * 8,
 list("PPPPPPPP"),
 list("RNBQKBNR")
 ]
 self.white_turn = True
def print_board(self):
   for row in self.board:
      print(' '.join(row))
   print()

def get_pawn_moves(self, r, c):
   moves = []
   p = self.board[r][c]
   dir = -1 if p.isupper() else 1
   start = 6 if p.isupper() else 1
   # Move forward
   if self.board[r + dir][c] == ' ':
      moves.append((r + dir, c))
      # Double move from start
      if r == start and self.board[r + 2 * dir][c] == ' ':
         moves.append((r + 2 * dir, c))
   # Captures
   for dc in [-1, 1]:
      nc = c + dc
      if 0 <= nc < 8:
         target = self.board[r + dir][nc]
         if target != ' ' and target.isupper() != p.isupper():
            moves.append((r + dir, nc))
   return moves

def is_valid_move(self, r, c, nr, nc):
   if not (0 <= nr < 8 and 0 <= nc < 8):
      return False
   if self.board[nr][nc].isupper() == self.white_turn:
      return False
   return (nr, nc) in self.get_pawn_moves(r, c)

def make_move(self, r, c, nr, nc):
   self.board[nr][nc], self.board[r][c] = self.board[r][c], ' '
   self.white_turn = not self.white_turn

def play(self):
  while True:
     self.print_board()
     print("White's" if self.white_turn else "Black's",
"move")
     s = input("From (e.g. a2): ")
     d = input("To (e.g. a4): ")
     sc, sr = ord(s[0]) - 97, 8 - int(s[1])
     dc, dr = ord(d[0]) - 97, 8 - int(d[1])
     if self.board[sr][sc].lower() != 'p' or not
self.is_valid_move(sr, sc, dr, dc):
        print("Invalid move")
        continue
     self.make_move(sr, sc, dr, dc)
ChessGame().play()
```

2. Implement an algorithm to find the shortest route and travel time between two locations within a city's transportation network.

```
iimport heapq
class Graph:
 def init(self):
 self.edges = {}
def add_edge(self, u, v, w):
   self.edges.setdefault(u, []).append((v, w))
   self.edges.setdefault(v, []).append((u, w))
def dijkstra(self, start):
   dist = {n: float('inf') for n in self.edges}
   dist[start] = 0
   prev, heap = {}, [(0, start)]
   while heap:
      d, u = heapq.heappop(heap)
      if d > dist[u]:
         continue
      for v, w in self.edges[u]:
         if d + w < dist[v]:
            dist[v], prev[v] = d + w, u
            heapq.heappush(heap, (dist[v], v))
   return dist, prev
def shortest_path(self, start, end):
   dist, prev = self.dijkstra(start)
   path = []
   while end:
      path.append(end)
      end = prev.get(end)
   return path[::-1], dist[path[0]]
def main():
 g = Graph()
 for _ in range(int(input("No. of paths: "))):
 u, v, w = input("from to time: ").split()
 g.add_edge(u, v, int(w))
 s, e = input("Start: "), input("End: ")
 path, time = g.shortest_path(s, e)
 print(" -> ".join(path), f"\nTime: {time}")
if name == "main":
 main()
```

5.
Implement Segment Tree for Range Sum Query in a Real-time Data Analytics Platform for student management system.

```
class SegmentTree:
 def init(self, arr):
 self.n = len(arr)
self.tree = [0] * (2 * self.n)
self.build(arr)
def build(self, arr):
   for i in range(self.n):
      self.tree[self.n + i] = arr[i]
   for i in range(self.n - 1, 0, -1):
      self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

def update(self, i, val):
   i += self.n
   self.tree[i] = val
   while i > 1:
      i //= 2
      self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

def query(self, l, r):
   l += self.n
   r += self.n
   res = 0
   while l <= r:
      if l % 2 == 1:
         res += self.tree[l]
         l += 1
      if r % 2 == 0:
         res += self.tree[r]
         r -= 1
     l //= 2
     r //= 2
   return res
arr = list(map(int, input("Array: ").split()))
st = SegmentTree(arr)
l, r = map(int, input("Query range (l r): ").split())
print("Sum:", st.query(l, r))
i, val = map(int, input("Update index and value: ").split())
st.update(i, val)
 print("Updated sum:", st.query(l, r))
```

3. Design a cost-efficient telecommunication network to connect multiple cities using Kruskal's algorithm

```
class DSU:
 def init(self, V):
self.p = {v: v for v in V}
self.r = {v: 0 for v in V}
def find(self, x):
   if self.p[x] != x:
      self.p[x] = self.find(self.p[x])
   return self.p[x]
def union(self, x, y):
   xr, yr = self.find(x), self.find(y)
   if xr == yr:
      return
   if self.r[xr] < self.r[yr]:
      self.p[xr] = yr
   elif self.r[xr] > self.r[yr]:
      self.p[yr] = xr
   else:
      self.p[yr] = xr
      self.r[xr] += 1
def kruskal(V, E):
 dsu, mst = DSU(V), []
 for u, v, w in sorted(E, key=lambda x: x[2]):
 if dsu.find(u) != dsu.find(v):
 dsu.union(u, v)
mst.append((u, v, w))
return mst
def main():
 V = input("Cities: ").split()
E = [tuple(input("city1 city2 cost: ").split()) for _ in
range(int(input("No. of connections: ")))]
mst = kruskal(V, [(u, v, int(w)) for u, v, w in E])
print("MST edges:")
 for u, v, w in mst:
 print(f"{u} - {v}: {w}")
if name == "main":
 main()
```

6. Implement Segment Tree for Range Sum Query in a Real-time Data Analytics Platform for student management system.

```
import numpy as np
class Node:
 def init(self, bounds):
self.b = bounds # (x, y, w, h)
self.cities = []
self.children = [None] * 4
class QuadTree:
def init(self, bounds, cap):
self.root = Node(bounds)
self.cap = cap
def insert(self, city):
   self._insert(self.root, city)
def _insert(self, node, city):
   if len(node.cities) < self.cap and node.children[0] is None:
      node.cities.append(city)
      return
   if node.children[0] is None:
      self._split(node)
   for child in node.children:
      if self._in_bounds(child.b, city):
         self._insert(child, city)
def _split(self, node):
   x, y, w, h = node.b
   hw, hh = w / 2, h / 2
   node.children = [
      Node((x, y, hw, hh)),
      Node((x + hw, y, hw, hh)),
      Node((x, y + hh, hw, hh)),
      Node((x + hw, y + hh, hw, hh)) ]
   for c in node.cities:
      for child in node.children:
         if self._in_bounds(child.b, c):
            self._insert(child, c)
   node.cities = []
def _in_bounds(self, b, p):
   x, y, w, h = b
   return x <= p[0] < x + w and y <= p[1] < y + h
def query(self, p):
   return self._query(self.root, p)
def _query(self, node, p):
   if node is None or not self._in_bounds(node.b, p):
      return []
   if node.children[0] is None:
      return node.cities
   result = []
   for ch in node.children:
      result += self._query(ch, p)
   return result
def nearest(self, p):
   return self._nearest(self.root, p, float('inf'), None)
def _nearest(self, node, p, best_dist, best):
   if node is None or not self._in_bounds(node.b, p):
      return best
   for c in node.cities:
      d = np.linalg.norm(np.array(p) - np.array(c))
      if d < best_dist:
         best_dist, best = d, c
   for ch in node.children:
      best = self._nearest(ch, p, best_dist, best)
   return best
---- Example usage ----
b = tuple(map(int, input("Boundary x y w h: ").split()))
cap = int(input("Capacity: "))
 qt = QuadTree(b, cap)
for _ in range(int(input("Number of cities: "))):
qt.insert(tuple(map(int, input("City x y: ").split())))
q = tuple(map(int, input("Query point x y: ").split()))
print("Cities nearby:", qt.query(q))
print("Nearest city:", qt.nearest(q))
```