LRU Cache

```
class Node:
   def __init__(self, key, val):
       self.key = key
       self.val = val
       self.prev = None
       self.next = None

class LRUCache:

   def __init__(self, capacity: int):
       self.cap = capacity
       self.cache = {}

       self.oldest = Node(0, 0)
       self.latest = Node(0, 0)
       self.oldest.next = self.latest
       self.latest.prev = self.oldest


   def get(self, key: int) -> int:
       if key in self.cache:
           self.remove(self.cache[key])
           self.insert(self.cache[key])
           return self.cache[key].val
       return -1

   def remove(self, node):
       prev, next = node.prev, node.next
       prev.next = next
       next.prev = prev

   def insert(self, node):
       prev, next = self.latest.prev, self.latest
       prev.next = next.prev = node
       node.next = next
       node.prev = prev

   def put(self, key: int, value: int) -> None:
       if key in self.cache:
```

```
            self.remove(self.cache[key])
        self.cache[key] = Node(key, value)
        self.insert(self.cache[key])

        if len(self.cache) > self.cap:
            lru = self.oldest.next
            self.remove(lru)
            del self.cache[lru.key]
```

Text Editor

```
class TextEditor:

    def __init__(self):
        self.left = []
        self.right = []

    def addText(self, text: str) -> None:
        for c in text:
            self.left.append(c)

    def deleteText(self, k: int) -> int:
        count = 0
        while len(self.left) > 0 and count < k:
            self.left.pop()
            count += 1
        return count

    def cursorLeft(self, k: int) -> str:
        count = 0
        while len(self.left) > 0 and count < k:
            self.right.append(self.left.pop())
            count += 1
        return ''.join(self.left[max(0, len(self.left) - 10):])

    def cursorRight(self, k: int) -> str:
        count = 0
        while len(self.right) > 0 and count < k:
            self.left.append(self.right.pop())
            count += 1
        return ''.join(self.left[max(0, len(self.left) - 10):])
```

robot

```
class Robot:
    def __init__(self):
        self.x = self.y = 0
        self.dir = 0  # 0:N, 1:E, 2:S, 3:W
        self.max_dist = 0
        self.dirs = [(0,1), (1,0), (0,-1), (-1,0)]

    def move(self, steps: int, obstacles: Set[Tuple[int,int]]):
        dx, dy = self.dirs[self.dir]
        for _ in range(steps):
            nx, ny = self.x + dx, self.y + dy
            if (nx, ny) in obstacles:
                break
            self.x, self.y = nx, ny
            self.max_dist = max(self.max_dist, self.x**2 + self.y**2)
def robotSim(commands: List[int], obstacles: List[List[int]]) -> int:
    obs = set(map(tuple, obstacles))
    robot = Robot()

    for cmd in commands:
        if cmd == -2:
            robot.dir = (robot.dir - 1) % 4
        elif cmd == -1:
            robot.dir = (robot.dir + 1) % 4
        else:
            robot.move(cmd, obs)

    return robot.max_dist
```

Paint house

```
def min_cost_colors(blue_costs, green_costs, red_costs):
    """
    Returns the sequence of colors ('b', 'g', 'r') with minimum total cost
    such that no two consecutive days have the same color.
    """
```

```
# Number of days
n = len(blue_costs)

# Costs indexed as: costs[color][day]
costs = [blue_costs, green_costs, red_costs]

# DP[i][c] = minimum cost up to day i if day i uses color c
DP = [[0] * 3 for _ in range(n)]

# parent[i][c] = color used on day i-1 that led to optimal DP[i][c]
parent = [[-1] * 3 for _ in range(n)]

# --- Day 0 initialization ---
for c in range(3):
    DP[0][c] = costs[c][0]

# --- Fill DP table ---
for i in range(1, n):
    for c in range(3):
        best_prev_cost = float("inf")
        best_prev_color = -1

        # Choose best previous color different from current
        for pc in range(3):
            if pc != c and DP[i - 1][pc] < best_prev_cost:
                best_prev_cost = DP[i - 1][pc]
                best_prev_color = pc

        DP[i][c] = costs[c][i] + best_prev_cost
        parent[i][c] = best_prev_color

# --- Find best final color ---
last_color = min(range(3), key=lambda c: DP[n - 1][c])

# --- Backtrack to get color sequence ---
answer = [0] * n
current = last_color

for i in range(n - 1, -1, -1):
    answer[i] = current
    current = parent[i][current]
```

```
    # Map indices to characters
    color_map = {0: 'b', 1: 'g', 2: 'r'}

    return [color_map[c] for c in answer]
```
```
        if costs is None:
            return 0
        n = len(costs)
        for i in range(1,n):
            costs[i][0] += min(costs[i-1][1], costs[i-1][2])
            costs[i][1] += min(costs[i-1][0], costs[i-1][2])
            costs[i][2] += min(costs[i-1][0], costs[i-1][1])
        return min(costs[-1])
```
URLS
```
class Codec:
    alphabet = string.ascii_letters + '0123456789'
    def __init__(self):
        self.url2code = {}
        self.code2url = {}

    def encode(self, longUrl):
        while longUrl not in self.url2code:
            code = ''.join(random.choice(Codec.alphabet) for _ in range(6))
            if code not in self.code2url:
                self.code2url[code] = longUrl
                self.url2code[longUrl] = code
        return 'http://tinyurl.com/' + self.url2code[longUrl]

    def decode(self, shortUrl):
        return self.code2url[shortUrl[-6:]]
```

File System

```
from collections import defaultdict
class Node:
    def __init__(self):
        self.child=defaultdict(Node)
        self.content=""

class FileSystem(object):

    def __init__(self):
        self.root=Node()

    def find(self,path):#find and return node at path.
        curr=self.root
        if len(path)==1:
            return self.root
        for word in path.split("/")[1:]:
            curr=curr.child[word]
        return curr

    def ls(self, path):
        curr=self.find(path)
        if curr.content:#file path,return file name
            return [path.split('/')[-1]]
        return sorted(curr.child.keys())

    def mkdir(self, path):
        self.find(path)

    def addContentToFile(self, filePath, content):
        curr=self.find(filePath)
        curr.content+=content

    def readContentFromFile(self, filePath):
        curr=self.find(filePath)
        return curr.content
```