Practice Solutions

SOI3011 Problem Solving Using Computational Thinking

Department of Data Science

Practice 1: Permutation

```
def permute_rec(a, l, r):
    if l == r:
        print(''.join(a))
    else:
        for i in range(l, r):
            a[l], a[i] = a[i], a[l]
            permute_rec(a, l+1, r)
            a[l], a[i] = a[i], a[l]
            # backtrack

def permute(s: str):
    a = list(s)
    permute_rec(a, 0, len(s))
```

- $O(n \times n!)$ time
- $O(n)$ for auxiliary space except output

Practice 1: Power Set

```
def power_set_recursive(A):
  if not A: return [[]]
  else:
    sets = power_set_recursive(A[:-1])
    newSets = []
    for curr in sets:
      new = curr.copy()
      new.append(A[-1])
      newSets.append(new)
    sets.extend(newSets)
    return sets
```

```
def power_set_iterative(A):
  sets = [[]]
  for n in A:
    newSets = []
    for curr in sets:
      new = curr.copy()
      new.append(n)
      newSets.append(new)
    sets.extend(newSets)
  return sets
```

- $O(n \times 2^n)$ time
- $O(n \times 2^n)$ space for output, so $O(1)$ for auxiliary space

Practice 2: Weave Linked List

```
def weave(head: Node):
  first = second = head
  while first:
    first = first.next.next
    second = second.next
  first = head
  while second:
    pointer = first.next
    first.next = second
    first = pointer
    pointer = second.next
    if not pointer:
      break
    second.next = first
    second = pointer
```

- Time complexity: $O(n)$
  n is the number of nodes in the linked list
- Space complexity: $O(1)$

Practice 2: Get Tree Level with Minimum Sum

```
def level_with_min_sum(node):
  if not node: return -1
  queue = deque([node])
  level, nCurrLevelNodes = 0, 1
  minSum, minSumLevel = float('inf'), -1
```

- $O(n)$ time
- $O(n)$ space where n is the number of nodes in a tree

```
  while queue:
    sumCurrLevel, nNextLevelNodes = 0, 0
    while nCurrLevelNodes:
      curr = queue.popleft()
      if curr.left:
        queue.append(curr.left)
        nNextLevelNodes += 1
      if curr.right:
        queue.append(curr.right)
        nNextLevelNodes += 1
      sumCurrLevel += curr.data
      nCurrLevelNodes -= 1
    if sumCurrLevel <= minSum:
      minSum = sumCurrLevel
      minSumLevel = level
    nCurrLevelNodes = nNextLevelNodes
    level += 1
  return minSumLevel
```

Practice 3: Cut Brick Wall

```
def fewest_number(bricks):
  edges = {}
  for row in bricks:
    length = 0
    for loc in row[:-1]:
      length += loc
      if length in edges:
        edges[length] += 1
      else:
        edges[length] = 1
  return len(bricks) - max(edges.values())
```

- Given a wall with length m, and n total bricks.
- $O(n)$ time
- $O(m)$ space

Practice 3: Find the Largest Subarray with 0 Sum

```
def longest_zero_sum_naive(arr):
  maxLen = 0
  for i, _ in enumerate(arr):
    currSum = 0
    for j in range(i, len(arr)):
      currSum += arr[j]
      if currSum == 0:
        maxLen = max(maxLen, j - i + 1)
  return maxLen
```

- $O(n^2)$ time
  where n is the number of elements in arr
- $O(1)$ space

```
def longest_zero_sum(arr):
  h = {}
  maxLen = 0
  prefixSum = 0
  for i, e in enumerate(arr):
    prefixSum += e
    if prefixSum in h:
      maxLen = max(maxLen, i - h[prefixSum])
    else:
      h[prefixSum] = I
  return maxLen
```

- $O(n)$ time
- $O(m)$ space

Practice 4: Compute Running Median

```
import heapq
def runningMedian(nums):
  left, right, output = [], [], []
  median = float('inf')
  for num in nums:
    inversed = False
    if num < median:
      h, g = left, right
      num = -num
      inversed = True
    else:
      h, g = right, left
```

- $O(nlogn)$ time where n is number of elems in nums
- $O(n)$ space

```
    lenH, lenG = len(h), len(g)
    if lenH == lenG:
      heapq.heappush(h, num)
      median = -h[0] if inversed else h[0]
    elif lenH > lenG:
      val = heapq.heappop(h)
      heapq.heappush(g, -val)
      heapq.heappush(h, num)
      x, y = (-h[0],g[0]) if inversed else (h[0],-g[0])
      median = (x + y) / 2
    else:
      heapq.heappush(h, num)
      x, y = (-h[0],g[0]) if inversed else (h[0],-g[0])
      median = (x + y) / 2
    output.append(median)
  return output
```

Practice 4: One Away

```
def oneAway(s1, s2):
  x, y = (s1, s2) if len(s1)<=len(s2) else (s2, s1)
  # Replace a character
  if len(x) == len(y):
    oneEdit = False
    for c1, c2 in zip(x, y):
      if c1 != c2:
        if oneEdit: return False
        else: oneEdit = True
    return True
```

- $O(n)$ time where n is the length of a string
- $O(1)$ space

```
# Insert or remove a character
 elif len(x) + 1 == len(y):
   i, j, oneEdit = 0, 0, False
   while i < len(x) and j < len(y):
     c1, c2 = x[i], y[j]
     if c1 == c2:
       i += 1
       j += 1
     else:
       if oneEdit: return False
       else:
         oneEdit = True
         j += 1
   return True
 else:
   return False
```

Practice 5: Bowling

- Subproblems. $B(i)$ = maximum score possible with pins $i, i+1, \dots, n-1$
- Original problem. $B(0)$
- Relate: $B(i) = \max(B(i+1),\ v_i + B(i+1), v_i \cdot v_{i+1} + B(i+2))$
- Topological order: decreasing i, for $i = n-1, n-2, \dots, 1, 0$
- Base case: $B(n) = 0$
- Time: computing maximum in one recurrence $B(i)$ takes $\Theta(1)$ time.
  Total running time is $\Theta(1) \times n = \Theta(n)$.

```
// Bottom-up pseudocode
B[n] = 0
for i = n-1, n-2, …, 1, 0
  B[i] = max(B[i+1], v[i] + B[i+1], v[i]*v[i+1] + B[i+2])
return B[0]
```

Bottom-Up Solution of Bowling

```
def bowlingBottomUp(nums: List[int]) -> int:
  n = len(nums)
  if n == 0: return 0
  ans = [0] * (n + 1)
  ans[n - 1] = max(0, nums[n - 1])
  for i in range(n - 2, -1, -1):
    ans[i] = max(ans[i+1], nums[i] + ans[i+1], nums[i] * nums[i+1] + ans[i+2])
  return ans[0]
```

- $O(n)$ time
- $O(n)$ space

Bottom-Up Solution + Constant Space

```
def bowling(nums: List[int]) -> int:
  n = len(nums)
  if n == 0: return 0
  oldPrev = 0
  latestPrev = max(0, nums[n - 1])
  for i in range(n - 2, -1, -1):
    curr = max(latestPrev, nums[i] + latestPrev, nums[i] * nums[i + 1] + oldPrev)
    oldPrev = latestPrev
    latestPrev = curr
  return lastestPrev
```

- $O(n)$ time
- $O(1)$ space

Practice 5: Robot in a Grid

```
def get_path(G, xs, ys, xt, yt, path, is_failed):
  r, c = len(G), len(G[0])
  cands = [(xs+1, ys), (xs, ys+1)]
  for xn, yn in cands:
    if xn >= r or yn >= c \
      or G[xn][yn] or (xn, yn) in is_failed:
      continue
    path.append((xn, yn))
    if (xn, yn) == (xt, yt):
      return path
    else:
      output = get_path(G, xn, yn, xt, yt, path, is_failed)
      if not output: is_failed.add((xn, yn))
      if output is not None: return output
    path.pop()
  is_failed.add((xs, ys))
  return None
```

```
def robot(grid):
  if not grid or not grid[0]:
    return []
  s = (0, 0)
  t = (len(grid)-1, len(grid[0])-1)
  if s == t:
    return [s]
  else:
    is_failed = set()
    return get_path(grid, *s, *t, [s], is_failed)
```

- $O(rc)$ time since we hit each cell just once.
- $O(rc)$ space
- r : the number of rows
- c : the number of columns

Lecture 4: Reconstruct Tree

```
def reconstruct_tree(preorder: list[str], inorder):
  def helper(l, n, prev):
    if n <= 0:
    return None
    val = preorder[l]
    curr = str2idx[val]
    n_left = curr - prev - 1 if curr > prev else n - prev + curr
    n_right = n - curr + prev if curr > prev else prev - curr - 1
    left = helper(l + 1, n_left, curr)
    right = helper(l + 1 + n_left, n_right, curr)
    return TreeNode(val, left, right)
  str2idx = {x: i for i, x in enumerate(inorder)}
  return helper(0, len(inorder), -1)
```

Practice 7: Compute Flight Itinerary

```
from collections import defaultdict

def itinerary(flights: list[tuple], start: str) ->
list[str]:

    flight_map = defaultdict(list); all_cities = set()

    for origin, dest in flights:

        flight_map[origin].append(dest)

        all_cities.add(origin); all_cities.add(dest)

    for origin in flight_map:

        flight_map[origin].sort()

    path = []; used_flights = set()

    visited = set([start])

    def visit(airport):

        while flight_map[airport]:

            next_airport = flight_map[airport].pop(0)

            flight = (airport, next_airport)

            if flight not in used_flights:

                used_flights.add(flight)

                visited.add(next_airport)

                visit(next_airport)

        path.insert(0, airport)
```

```
    visit(start)

    if len(visited) == len(all_cities):

        return path

    else:

        return None
```

$O(n!)$ time where n is the number of cities
$O(n+m)$ space where m is the number of flights

Practice 7: Combination Sum

```
def backtrack(candidates, idx, target, intermediate, results):
  for i in range(idx, len(candidates)):
    cand = candidates[i]
    if i > idx and cand == candidates[i-1]: continue
    if target == cand:
      intermediate.append(cand)
      results.append(intermediate.copy())
      intermediate.pop()
    elif target > cand:
      intermediate.append(cand)
      backtrack(candidates, i + 1, target - cand, intermediate, results)
      intermediate.pop()


def combinationSum2(candidates: List[int], target: int) -> List[List[int]]:
  results = []
  candidates.sort()
  backtrack(candidates, 0, target, [], results)
  return results
```

- $O(k \cdot 2^n)$ where k is the average length of each solution.
- $O(n)$ auxiliary space.