Repository Code Export

Generated from: `C:\Users\Shu_Pang\Desktop\my_assignment`

File: export.py

```python
import os

def pack_python_to_markdown(root_directory, output_file):
    with open(output_file, 'w', encoding='utf-8') as md_file:
        md_file.write(f"# Repository Code Export\n\n")
        md_file.write(f"Generated from: `{os.path.abspath(root_directory)}`\n\n---
\n")

        # Walk through the directory and all subdirectories
        for root, dirs, files in os.walk(root_directory):
            # Optional: Skip hidden folders or virtual environments
            dirs[:] = [d for d in dirs if not d.startswith('.') and d !=
'__pycache__']

            for file in files:
                if file.endswith(".py"):
                    # Get the full path of the file
                    full_path = os.path.join(root, file)
                    # Get relative path for the header
                    relative_path = os.path.relpath(full_path, root_directory)

                    print(f"Packing: {relative_path}")

                    # Write the header to Markdown
                    md_file.write(f"## File: {relative_path}\n")
                    md_file.write("```python\n")

                    # Read the content of the python file
                    try:
                        with open(full_path, 'r', encoding='utf-8') as f:
                            md_file.write(f.read())
                    except Exception as e:
                        md_file.write(f"# Error reading file: {e}")

                    md_file.write("\n```\n\n---\n")

    print(f"\nSuccess! All code packed into: {output_file}")

if __name__ == "__main__":
    # SETTINGS: Change '.' to your folder path if needed
    target_folder = "."
    output_name = "full_repository_code.md"
```

```
    pack_python_to_markdown(target_folder, output_name)
```

File: 2025_fall_python_assignment\assignmentC\bfs_maze.py

```python
#given a n*m maze with walls(1) and paths(0), find the smallest number of steps
#from the top_left corner to every other cell using BFS
#if it is not possible to reach a cell, mark it as -1, return the resulting grid
from collections import deque
def bfs_maze(maze, n, m):
    # Initialize the result grid with -1
    result = [[-1 for _ in range(m)] for _ in range(n)]

    # Directions for moving up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # BFS initialization
    queue = deque()

    # Start from the top-left corner if it's a path
    if maze[0][0] == 0:
        queue.append((0, 0))
        result[0][0] = 0

    while queue:
        x, y = queue.popleft()

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            # Check if the new position is within bounds and is a path
            if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] == 0:
                # If the cell has not been visited yet
                if result[nx][ny] == -1:
                    result[nx][ny] = result[x][y] + 1
                    queue.append((nx, ny))

    return result

n, m = map(int, input().split())
maze = [list(map(int, input().split())) for _ in range(n)]
result = bfs_maze(maze, n, m)
for row in result:
    print(' '.join(map(str, row)))
```

File: 2025_fall_python_assignment\assignmentC\test1.py

```
length, nums = map(int, input().split())
length += 1
interval = []
for _ in range(nums):
    interval.append(tuple(map(int, input().split())))
interval.sort()
previous_end = -1
for start, end in interval:
    if start >= previous_end:
        length -= (end - start if start == previous_end else end - start + 1)
        previous_end = end
    elif start < previous_end < end:
        length -= (end - previous_end)
        previous_end = end
    else:
        continue
print(length)
```

File: 2025_fall_python_assignment\assignmentC\test10.py

```
# this time we repeat test 9 using dfs and backtracking
def calcu_methods(string : str, target : int) -> int:
    '''please be merciful on test case since you give no restraint on it'''
    # start dfs in no time
    count = 0
    n = len(string)
    def dfs(index, current_value, last_operand):
        nonlocal count
        if index == n:
            if current_value == target:
                count += 1
            return
        else:
            for i in range(index, n):
                if i > index and string[index] == "0":
                    break
                else:
                    part = string[index:i+1]
                    operand = int(part)

                if index == 0:
                    dfs(i+1, operand, operand)
                else:
                    # +
                    dfs(i+1, current_value + operand, operand)
                    # -
                    dfs(i+1, current_value - operand, -operand)
                    # *
                    dfs(i+1, current_value - last_operand + last_operand *
operand, last_operand * operand)
    dfs(0, 0, 0)
    return count
string = str(input().strip())
target = int(input().strip())
result = calcu_methods(string, target)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\test2.py

```
def generate_message(message, phone_number) -> bool:
    dict_words = {2: ["a", "b", "c"],
                  3: ["d", "e", "f"],
                  4: ["g", "h", "i"],
                  5: ["j", "k", "l"],
                  6: ["m", "n", "o"],
                  7: ["p", "q", "r", "s"],
                  8: ["t", "u", "v"],
                  9: ["w", "x", "y", "z"]}
    if len(message) != len(phone_number):
        return False
    message = message.lower()
    number_lis = list(phone_number)
    number_lis = [int(i) for i in number_lis]
    if 0 in number_lis or 1 in number_lis:
        return False
    for i in range(len(message)):
        if message[i] in dict_words[number_lis[i]]:
            continue
        else:
            return False
    return True

cases = int(input())
for _ in range(cases):
    msg, num = map(str, input().split())
    if generate_message(msg, num):
        print("Y")
    else:
        print("N")
```

File: 2025_fall_python_assignment\assignmentC\test3.py

```
# matrix will be used to store the values
m, n = map(int, input().split())
```

```
matrix = []
for i in range(m):
    row = list(map(int, input().split()))
    matrix.append(row)
most_one_row = 0
current_max = 0
for i in range(m):
    count_i_row = sum(matrix[i])
    if count_i_row > current_max:
        current_max = count_i_row
        most_one_row = i
print(most_one_row)
print(current_max)
```

File: 2025_fall_python_assignment\assignmentC\test4.py

```
candies = list(map(int, input().split()))
extra = int(input())
max_candies = max(candies)
for i in range(len(candies)):
    if candies[i] == max_candies:
        print(1)
        continue
    else:
        if candies[i] + extra >= max_candies:
            print(1)
        else:
            print(0)
```

File: 2025_fall_python_assignment\assignmentC\test5.py

```
# numbers that have nothing to do with 7
def unrelated_to_seven(n) -> int:
    no_related = 0
    for i in range(1, n + 1):
        if '7' not in str(i) and i % 7 != 0:
            no_related += i**2
    return no_related

n = int(input())
print(unrelated_to_seven(n))
```

File: 2025_fall_python_assignment\assignmentC\test6.py

```python
# simulate the accidents happening in a time ordered manner and find the smallest
number of chair needed
def min_chairs(accidents):
    min_chairs_needed = 0
    current_chairs_needed = 0
    for i in accidents:
        if i == "E":
            current_chairs_needed += 1
            min_chairs_needed = max(min_chairs_needed, current_chairs_needed)
        elif i == "L":
            current_chairs_needed -= 1
    return min_chairs_needed

accident = input().strip()
print(min_chairs(accident))
```

File: 2025_fall_python_assignment\assignmentC\test7.py

```python
# use a matrix to decide who is the strongest player
total_players = int(input())
matrix = []
for _ in range(total_players):
    row = list(map(int, input().split()))
    matrix.append(row)
win_list = set()
lost_list = set()
for i in range(total_players):
    for j in range(total_players):
        if i == j:
            continue
        if i in lost_list and j in lost_list:
            continue
        if matrix[i][j] == 1:
            win_list.add(i)
            lost_list.add(j)
            if j in win_list:
                win_list.remove(j)
        else:
            win_list.add(j)
            lost_list.add(i)
            if i in win_list:
                win_list.remove(i)
if len(win_list) == 1:
    print(win_list.pop())
```

File: 2025_fall_python_assignment\assignmentC\test8.py

```python
# use dfs to find the number of islands
from collections import deque
def calc_islands(m, n, matrix):
    visited = [[False for _ in range(n)] for _ in range(m)]
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    count_islands = 0
    for x in range(m):
        for y in range(n):
            if matrix[x][y] == 0:
                visited[x][y] = True
                continue
            if visited[x][y]:
                continue
            else:
                visited[x][y] = True
                count_islands += 1
                stack = deque()
                stack.append((x, y))
                while stack:
                    cur_x, cur_y = stack.popleft()
                    for dx, dy in directions:
                        new_x, new_y = cur_x + dx, cur_y + dy
                        if 0 <= new_x < m and 0 <= new_y < n and not
visited[new_x][new_y] and matrix[new_x][new_y] == 1:
                            visited[new_x][new_y] = True
                            stack.append((new_x, new_y))
    return count_islands

m, n = map(int, input().split())
matrix = []
for _ in range(m):
    row = list(map(int, input().split()))
    matrix.append(row)
result = calc_islands(m, n, matrix)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\test9.py

```python
"""Given a numeric string, insert the operators +, - and * between digits
(or join digits) to form expressions that evaluate to the target value.

This module provides `count_expressions(num_str, target)` which returns
the number of distinct expressions that evaluate to `target`.

Example:
    num_str = "123", target = 6 -> 2 ways: "1+2+3" and "1*2*3"
"""

from typing import List
```

```
def count_expressions(num_str: str, target: int) -> int:
    """Return the number of ways to insert +, -, * to get `target`.

    Args:
        num_str: string consisting of digits '0'-'9'.
        target: integer target value.

    Returns:
        Integer count of distinct valid expressions.
    """

    n = len(num_str)
    if n == 0:
        return 0

    count = 0

    def dfs(index: int, current_value: int, last_operand: int) -> None:
        nonlocal count
        if index == n:
            if current_value == target:
                count += 1
            return

        # Try all possible next numbers by extending the substring
        for i in range(index, n):
            # Avoid numbers with leading zeros (like "05") except single '0'
            if i > index and num_str[index] == '0':
                break
            part = num_str[index : i + 1]
            operand = int(part)

            if index == 0:
                # First number in the expression — it sets the running value
                dfs(i + 1, operand, operand)
            else:
                # '+' operator
                dfs(i + 1, current_value + operand, operand)
                # '-' operator
                dfs(i + 1, current_value - operand, -operand)
                # '*' operator: undo last_operand and apply multiplication
                dfs(i + 1, current_value - last_operand + last_operand * operand,
last_operand * operand)

    dfs(0, 0, 0)
    return count

string = str(input().strip())
target = int(input().strip())
result = count_expressions(string, target)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\the_first_problem_to_solve.py

```python
#need to place pieces on a board of a given shape
#the shape is given in a matrix, where # represents a grid cell that can be occupied, and . represents it cannot be occupied
#any two pieces cannot be placed in the same row or column, figure out how many ways there are to place the pieces
#multiple test cases, each starts with n, meaning the size of the board (n x n) and k, meaning the number of pieces to place
#followed by n lines, each containing n characters (# or .)
#-1 -1 means end of input
def count_ways(board, n, k, row=0, placed=0, cols=set()):
    if placed == k:
        return 1
    if row == n:
        return 0

    total_ways = 0
    for col in range(n):
        if board[row][col] == '#' and col not in cols:
            cols.add(col)
            total_ways += count_ways(board, n, k, row + 1, placed + 1, cols)
            cols.remove(col)

    total_ways += count_ways(board, n, k, row + 1, placed, cols)
    return total_ways

def main():
    while True:
        n, k = map(int, input().split())
        if n == -1 and k == -1:
            break
        board = [input().strip() for _ in range(n)]
        result = count_ways(board, n, k)
        print(result)

main()
```

File: 2025_fall_python_assignment\assignmentC\the_first_project.py

```python
print("Hello to be here.")
```

File: 2025_fall_python_assignment\assignmentC\random_practice\boredom.py

```
#give you a sequence of integers, you can choose a number i in sequence and delete
it,
#then all i+1 and i-1 in this sequence will also be deleted, you can get i points
for deleting i,
#from the remaining sequence you can continue to do this operation, what is the
maximum points you can get
def max_points(sequence):
    from collections import Counter

    if not sequence:
        return 0

    count = Counter(sequence)
    max_num = max(count)

    dp = [0] * (max_num + 1)
    dp[0] = 0
    dp[1] = count[1] * 1

    for i in range(2, max_num + 1):
        dp[i] = max(dp[i - 1], dp[i - 2] + count[i] * i)

    return dp[max_num]

n = int(input())
sequence = list(map(int, input().split()))
result = max_points(sequence)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\random_practice\choose_the_player.py

```
# given two sequence of player heights, the players were numbered from 1 to n
# we need to choose players such that no two chosen players are adjacent and the
number of chosen players should strictly increase
# return the maximum height sum of the chosen players
def choose_method(n, height1, height2) -> int:
    dp = [[0] * 3 for _ in range(n + 1)]

    for i in range(1, n + 1):
        h1 = height1[i - 1]
        h2 = height2[i - 1]
        dp[i][0] = max(dp[i - 1])  # not choosing any player at position i
        dp[i][1] = max(dp[i - 1][0] + h1, dp[i-1][2] + h1)  # choosing player from
height1
        dp[i][2] = max(dp[i - 1][0] + h2, dp[i-1][1] + h2)  # choosing player from
height2

    return max(dp[n])

n = int(input())
height1 = list(map(int, input().split()))
height2 = list(map(int, input().split()))
result = choose_method(n, height1, height2)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\random_practice\how_wealthy_people_buy.py

```
#Given a sequence of integers, where each number represents the value of a
commodity (which can be negative). The rich man's method of buying is 'I will take
all the commodities from the nth to the kth!' (n<=k). In other words, the rich man
will definitely buy several consecutive commodities. After buying, the rich man
will decide based on his mood to put back at most one commodity (he can choose not
to return it). However, the rich man cannot return empty-handed; he must take at
least one commodity. How much is the maximum total value of the commodities the
smart (?) rich man can buy?
#use dp to solve this
#nth isn't necessarily the first item in the sequence, it can be any item in the
sequence
"""Given a sequence of integers, where each number represents the value of a
commodity (which can be negative). The rich man's method of buying is 'I will take
all the commodities from the nth to the kth!' (n<=k). In other words, the rich man
will definitely buy several consecutive commodities. After buying, the rich man
will decide based on his mood to put back at most one commodity (he can choose not
to return it). However, the rich man cannot return empty-handed; he must take at
least one commodity. How much is the maximum total value of the commodities the
smart (?) rich man can buy?
use dp to solve this
nth isn't necessarily the first item in the sequence, it can be any item in the
sequence
"""


def max_wealthy_purchase(sequence):
    n = len(sequence)
    if n == 0:
        return 0

    # Initialize dp arrays
    dp_no_return = [0] * n  # Max sum ending at i without returning any item
    dp_with_return = [float('-inf')] * n  # Max sum ending at i with returning one
```

```
item

    dp_no_return[0] = sequence[0]

    max_value = sequence[0]

    for i in range(1, n):
        dp_no_return[i] = max(dp_no_return[i - 1] + sequence[i], sequence[i])
        dp_with_return[i] = max(dp_with_return[i - 1] + sequence[i],
dp_no_return[i - 1])

        max_value = max(max_value, dp_no_return[i], dp_with_return[i])

    return max_value


if __name__ == '__main__':
    sequence = list(map(int, input().split(",")))
    result = max_wealthy_purchase(sequence)
    print(result)
```

File: 2025_fall_python_assignment\assignmentC\random_practice\maze_easiest.py

```
from collections import deque
def find_smallest_steps(maze, n, m):
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    q = deque()
    steps = [[-1 for _ in range(m)] for _ in range(n)]
    steps[0][0] = 0
    q.append((0, 0))
    while q:
        x, y = q.popleft()
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] == 0:
                if steps[nx][ny] == -1:
                    steps[nx][ny] = steps[x][y] + 1
                    q.append((nx, ny))
    return steps[n-1][m-1]

n, m = map(int, input().split())
maze = [list(map(int, input().split())) for _ in range(n)]
result = find_smallest_steps(maze, n, m)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\random_practice\number_operation.py

```python
#from the integer 1, you can perform the following operations:
# multiply by 2
# plus 1
# Given a target number n, find the minimum number of operations to reach n from 1
using bfs
from collections import deque
def min_operations(n):
    if n == 1:
        return 0

    queue = deque()
    queue.append((1, 0))  # (current number, steps)
    visited = set()
    visited.add(1)

    while queue:
        current, steps = queue.popleft()

        # Generate the next possible numbers
        next_nums = [current * 2, current + 1]

        for next_num in next_nums:
            if next_num == n:
                return steps + 1
            if next_num < n * 2 and next_num not in visited:  # limit search space
                visited.add(next_num)
                queue.append((next_num, steps + 1))

    return -1  # should not reach here for valid n

n = int(input())
result = min_operations(n)
print(result)
```

File: 2025_fall_python_assignment\assignmentC\random_practice\pots.py

```python
def fill_the_pot(a, b, c):
    """using fill and drop and pour to get desired amount of water"""
    from collections import deque
    visited = set()
    queue = deque()
    queue.append((0, 0))
    parent = {}
    while queue:
        x, y = queue.popleft()
```

```
            if (x, y) in visited:
                continue
            visited.add((x, y))

            if x == c or y == c:
                path = []
                while (x, y) in parent:
                    px, py, action = parent[(x, y)]
                    path.append(action)
                    x, y = px, py
                path.reverse()
                print(len(path))
                for step in path:
                    print(step)
                return

            # FILL pot 1
            if (a, y) not in visited:
                queue.append((a, y))
                parent[(a, y)] = (x, y, "FILL(1)")

            # FILL pot 2
            if (x, b) not in visited:
                queue.append((x, b))
                parent[(x, b)] = (x, y, "FILL(2)")

            # DROP pot 1
            if (0, y) not in visited:
                queue.append((0, y))
                parent[(0, y)] = (x, y, "DROP(1)")

            # DROP pot 2
            if (x, 0) not in visited:
                queue.append((x, 0))
                parent[(x, 0)] = (x, y, "DROP(2)")

            # POUR from pot 1 to pot 2 (both fit)
            if x + y <= b and (0, x + y) not in visited:
                queue.append((0, x + y))
                parent[(0, x + y)] = (x, y, "POUR(1,2)")

            # POUR from pot 2 to pot 1 (both fit)
            if x + y <= a and (x + y, 0) not in visited:
                queue.append((x + y, 0))
                parent[(x + y, 0)] = (x, y, "POUR(2,1)")

            # POUR from pot 1 to pot 2 (pot 2 fills up)
            if x + y > b and (x - (b - y), b) not in visited:
                queue.append((x - (b - y), b))
                parent[(x - (b - y), b)] = (x, y, "POUR(1,2)")

            # POUR from pot 2 to pot 1 (pot 1 fills up)
            if x + y > a and (a, y - (a - x)) not in visited:
                queue.append((a, y - (a - x)))
                parent[(a, y - (a - x))] = (x, y, "POUR(2,1)")

        print("impossible")
        return

    # Read input
    a, b, c = map(int, input().split())
    fill_the_pot(a, b, c)
```

File: 2025_fall_python_assignment\assignmentC\random_practice\vacation.py

```
n = int(input())
state_lis = list(map(int, input().split()))
def sleep_when_old(n: int, state_lis: list) -> int:
    """Return the minimum number of rest days.

    dp_table[i][0] = min rest days up to day i if day i is rest
    dp_table[i][1] = ... if day i is gym
    dp_table[i][2] = ... if day i is contest
    """
    # make the list 1-indexed
    state_lis.insert(0, 0)
    INF = float('inf')

    # proper independent rows and initialize with INF
    dp_table = [[INF] * 3 for _ in range(n + 1)]
    dp_table[0] = [0, 0, 0]

    for i in range(1, n + 1):
        # rest: take the best of previous day and add 1 rest day
        dp_table[i][0] = min(dp_table[i - 1]) + 1

        s = state_lis[i]
        if s == 0:
            dp_table[i][1] = INF
            dp_table[i][2] = INF
        elif s == 2:  # gym only
            dp_table[i][1] = min(dp_table[i - 1][0], dp_table[i - 1][2])
            dp_table[i][2] = INF
        elif s == 1:  # contest only
            dp_table[i][1] = INF
            dp_table[i][2] = min(dp_table[i - 1][0], dp_table[i - 1][1])
        else:  # s == 3, both available
            dp_table[i][1] = min(dp_table[i - 1][0], dp_table[i - 1][2])
            dp_table[i][2] = min(dp_table[i - 1][0], dp_table[i - 1][1])

    return int(min(dp_table[n]))
result = sleep_when_old(n, state_lis)
print(result)
```

File: 2025_fall_python_assignment\assignmentD\counterfeit_coin.py

```

```

File: 2025_fall_python_assignment\assignmentD\distance_on_editing.py

```
# give you word1 and word2, find the minimum number of operations required to convert word1 to word2
# operations are insert a character, delete a character, replace a character
def min_distance(word1: str, word2: str) -> int:
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(
                    dp[i - 1][j] + 1,    # delete
                    dp[i][j - 1] + 1,    # insert
                    dp[i - 1][j - 1] + 1  # replace
                )

    return dp[m][n]
```

File: 2025_fall_python_assignment\assignmentD\plus_sequence.py

```
# give you a sequence, find the subsequence with the largest multiplicative sum and return the sum
# dp required
# the start point don't need to be the first element
def max_product_subsequence(seq):
    if not seq:
        return 0
    n = len(seq)
    max_product = [0] * n
    min_product = [0] * n

    max_product[0] = seq[0]
    min_product[0] = seq[0]
    result = seq[0]

    for i in range(1, n):
        if seq[i] > 0:
            max_product[i] = max(seq[i], max_product[i - 1] * seq[i])
            min_product[i] = min(seq[i], min_product[i - 1] * seq[i])
        else:
            max_product[i] = max(seq[i], min_product[i - 1] * seq[i])
            min_product[i] = min(seq[i], max_product[i - 1] * seq[i])

        result = max(result, max_product[i])

    return result
```

File: 2025_fall_python_assignment\assignmentD\put_words_together.py

```
# give you a string and a word dict, determine if the string can be formed by
words in the dict
# you can use one word multiple times
def word_break(s: str, word_dict: set) -> bool:
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True  # empty string can be formed

    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in word_dict:
                dp[i] = True
                break

    return dp[n]
```

File: 2025_fall_python_assignment\assignmentD\radar_installation.py

```
def radar_optimize(lis, coverage) -> int:
    lis.sort()
    installed_intervals = []
    for x, y in lis:
        if y > coverage:
            return -1
        left = x - (coverage**2 - y**2)**0.5
        right = x + (coverage**2 - y**2)**0.5
        installed_intervals.append((left, right))
    installed_intervals.sort()
    count = 0
    current_end = -float('inf')
    for i, (left, right) in enumerate(installed_intervals):
        if left > current_end:
            count += 1
            current_end = right
        else:
            current_end = min(current_end, right)
    return count

case = 1
while True:
    n, d = map(int, input().strip().split())
    if n == 0 and d == 0:
        break
    islands = []
    for _ in range(n):
        x, y = map(int, input().strip().split())
        islands.append((x, y))

    result = radar_optimize(islands, d)
    print(f"Case {case}: {result}")
    case += 1
    cache = input()
```

File: 2025_fall_python_assignment\assignmentD\snoker.py

```
white_position = tuple(map(int, input().split()))
black_position = tuple(map(int, input().split()))
directions = tuple(map(int, input().split()))
energy = int(input().strip())
active = False
lis = [(0,0), (8, 0), (16, 0), (0, 5), (8, 5), (16, 5)]
set_of_the_holes = set(lis)
while energy > 0:
    dx, dy = directions
    white_position = (white_position[0] + dx, white_position[1] + dy)
    energy -= 1
    if white_position == black_position:
        active = True
        break
    if white_position in set_of_the_holes:
        print(-1)
        break
    if white_position[0] == 0 or white_position[0] == 16:
        directions = (-directions[0], directions[1])
    if white_position[1] == 0 or white_position[1] == 5:
        directions = (directions[0], -directions[1])

if active:
    while energy > 0:
        dx, dy = directions
        black_position = (black_position[0] + dx, black_position[1] + dy)
        energy -= 1
        if black_position in set_of_the_holes:
            print(1)
            exit(0)
            break
        if black_position[0] == 0 or black_position[0] == 16:
            directions = (-directions[0], directions[1])
        if black_position[1] == 0 or black_position[1] == 5:
            directions = (directions[0], -directions[1])

if energy == 0:
    print(0)
```

File: 2025_fall_python_assignment\assignmentD\square_number.py

```
# give you a integer, return the smallest number of number of perfect square numbers that sum to the integer
# you can use one perfect square number multiple times
def num_squares(n: int) -> int:
    dp = [float('inf')] * (n + 1)
    dp[0] = 0

    for i in range(1, n + 1):
        j = 1
        while j * j <= i:
            dp[i] = min(dp[i], dp[i - j * j] + 1)
            j += 1

    return dp[n]
```

File: 2025_fall_python_assignment\assignmentD\subway.py

```
# give you coordinates of your home and school, you walk at a speed of 10km per hour, one unit stands for 1m.
# you can take subway at certain stations, subway speed is 40km per hour.
# given the coordinates of subway stations, find the shortest time to get to school.
# you can switch between walking and subway at any subway station.
# there are different subway lines, stations will be given resp
```

```
import math
from typing import List, Tuple
def calculate_time(distance: float, speed_kmh: float) -> float:
    speed_mps = speed_kmh * 1000 / 3600  # convert km/h to m/s
    return distance / speed_mps
def shortest_time(home: Tuple[int, int], school: Tuple[int, int], stations: 
List[Tuple[int, int]]) -> float:
    points = [home] + stations + [school]
    n = len(points)
    dist = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i != j:
                distance = math.sqrt((points[i][0] - points[j][0]) ** 2 + 
(points[i][1] - points[j][1]) ** 2)
                walk_time = calculate_time(distance, 10)
                subway_time = calculate_time(distance, 40)
                dist[i][j] = min(walk_time, subway_time) if i != 0 and j != n - 1 
else walk_time
    dp = [float('inf')] * n
    dp[0] = 0
    for i in range(n):
        for j in range(i + 1, n):
            dp[j] = min(dp[j], dp[i] + dist[i][j])
    return dp[-1]
home_x, home_y = map(int, input().strip().split())
school_x, school_y = map(int, input().strip().split())
num_stations = int(input().strip())
stations = []
for _ in range(num_stations):
    station_x, station_y = map(int, input().strip().split())
    stations.append((station_x, station_y))

result = shortest_time((home_x, home_y), (school_x, school_y), stations)
print(round(result, 6))
```

File: 2025_fall_python_assignment\assignmentD\water_the_plants.py

```
n, a, b = map(int, input().strip().split())
current_a = a
current_b = b
plants_lis = list(map(int, input().strip().split()))
pointer_a = 0
pointer_b = n - 1
fill_time = 0
if n % 2 == 0:
    while pointer_a < pointer_b:
        if current_a < plants_lis[pointer_a]:
            fill_time += 1
            current_a = a - plants_lis[pointer_a]
        else:
            current_a -= plants_lis[pointer_a]
        if current_b < plants_lis[pointer_b]:
            fill_time += 1
            current_b = b - plants_lis[pointer_b]
        else:
            current_b -= plants_lis[pointer_b]
        pointer_a += 1
        pointer_b -= 1
    print(fill_time)
else:
    while pointer_a < pointer_b:
        if current_a < plants_lis[pointer_a]:
            fill_time += 1
            current_a = a - plants_lis[pointer_a]
        else:
            current_a -= plants_lis[pointer_a]
        if current_b < plants_lis[pointer_b]:
            fill_time += 1
            current_b = b - plants_lis[pointer_b]
        else:
            current_b -= plants_lis[pointer_b]
        pointer_a += 1
        pointer_b -= 1
    if pointer_a == pointer_b:
        if current_a >= plants_lis[pointer_a] or current_b >=
plants_lis[pointer_b]:
            pass
        else:
            fill_time += 1
    print(fill_time)
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test1.py

```
# find all mountains of one sequence
def find_mountains(seq, n):
    count = 0
    for i in range(1, n-1):
        if seq[i-1] < seq[i] > seq[i+1]:
            count += 1
    return count

n = int(input().strip())
sequence = list(map(int, input().strip().split()))
result = find_mountains(sequence, n)
print(result)
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test2.py

```
# find the longest mutual prefix in a str sequence
def longest_common_prefix(strs):
    if not strs:
        return ""

    prefix = strs[0]

    for s in strs[1:]:
        while not s.startswith(prefix):
            prefix = prefix[:-1]
            if not prefix:
                return ""

    return prefix

n = int(input().strip())
sequence = list(map(str, input().strip().split()))
print(longest_common_prefix(sequence))
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test3.py

```
def find_the_longest_one(lis, n) -> int:
    current_length = 0
    max_length = 0
    for i in range(n):
        if lis[i] == 1:
            current_length += 1
            max_length = max(max_length, current_length)
        else:
            current_length = 0
    return max_length

n = int(input().strip())
sequence = list(map(int, input().strip().split()))
print(find_the_longest_one(sequence, n))
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test4.py

```
def snake_in_matrix(n, operations) -> int:
    current_place = [0, 0]
    for op in operations:
        if op == "UP":
            current_place[0] -= 1
        elif op == "DOWN":
            current_place[0] += 1
        elif op == "LEFT":
            current_place[1] -= 1
        elif op == "RIGHT":
            current_place[1] += 1
    return (current_place[0] * n) + current_place[1]

n = int(input().strip())
operations = list(map(str, input().strip().split()))
print(snake_in_matrix(n, operations))
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test5.py

```
# you know stock prices in n days, find the max profit you can make, you can only sell once after buy once
def max_profit(prices, n) -> int:
    min_price = float('inf')
    max_profit = 0

    for price in prices:
        if price < min_price:
            min_price = price
        elif price - min_price > max_profit:
            max_profit = price - min_price

    return max_profit

n = int(input().strip())
prices = list(map(int, input().strip().split()))
print(max_profit(prices, n))
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test6.py

```
def judge_func(str):
    standard = "qwertyuiopasdfghjklzxcvbnm"
    lis = list(standard)
    bucket = []
    chr_set = set(lis)
    for char in str:
        if char in chr_set:
            bucket.append(char)
    length = len(bucket)
    if length == 0:
        return True
    elif length % 2 == 0:
        pointer1 = length // 2 - 1
        pointer2 = length // 2
        while pointer1 >= 0 and pointer2 < length:
            if bucket[pointer1] != bucket[pointer2]:
                return False
            pointer1 -= 1
            pointer2 += 1
        return True
    else:
        left = length // 2 - 1
        right = length // 2 + 1
        while left >= 0 and right < length:
            if bucket[left] != bucket[right]:
                return False
            left -= 1
            right += 1
        return True

string = str(input().strip())
if judge_func(string):
    print("True")
else:
    print("False")
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test7.py

```
# give you a matrix with some islands represented as 1s and water as 0s
# your task is to find the perimeter of the islands
def island_perimeter(grid) -> int:
    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])
    perimeter = 0

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 1:
                # Each land cell contributes 4 to the perimeter
                perimeter += 4
                # Check for adjacent land cells to subtract shared edges
                if r > 0 and grid[r - 1][c] == 1:  # Up
                    perimeter -= 2
                if c > 0 and grid[r][c - 1] == 1:  # Left
                    perimeter -= 2

    return perimeter

row, col = map(int, input().strip().split())
grid = []
for _ in range(row):
    grid.append(list(map(int, input().strip().split())))
print(island_perimeter(grid))
```

File: 2025_fall_python_assignment\assignmentD\mocking_test\test8.py

```
# give you a string of parentheses, find the longest substring that is valid
def longest_valid_parentheses(s: str) -> int:
    max_length = 0
    stack = [-1]

    for i, char in enumerate(s):
        if char == '(':
            stack.append(i)
        else:
            stack.pop()
            if not stack:
                stack.append(i)
            else:
                max_length = max(max_length, i - stack[-1])

    return max_length

str = input().strip()
print(longest_valid_parentheses(str))
```

File: 2025_fall_python_assignment\assignmentD\month_test\test1.py

```
n = int(input())
if n == 1:
    print("End")
else:
    while n != 1:
        if n % 2 == 0:
            n = n // 2
            print(f"{n*2}/2={n}")
        else:
            n = 3 * n + 1
            print(f"{(n-1)//3}*3+1={n}")
    print("End")
```

File: 2025_fall_python_assignment\assignmentD\month_test\test2.py

```
# give you a big integer string, and the number of numbers you will delete, find
the smallest possible number string after deletion
def remove_k_digits(num: str, k: int) -> str:
    stack = []
    for digit in num:
        while k > 0 and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    final_stack = stack[:-k] if k else stack
    result = ''.join(final_stack).lstrip('0')
    return result if result else '0'

num = str(input().strip())
k = int(input().strip())
print(remove_k_digits(num, k))
```

File: 2025_fall_python_assignment\assignmentD\month_test\test3.py

```
l = int(input())
n = int(input())
stu_lis = list(map(int, input().strip().split()))
max_time = max(max(i, l-i+1) for i in stu_lis)
min_time = max(min(i, l-i+1) for i in stu_lis)
print(min_time, max_time)
```

File: 2025_fall_python_assignment\assignmentD\month_test\test4.py

```
from collections import deque

cipher = list(input().strip())
cipher_list = []
alpha_lis = [chr(i) for i in range(ord('a'), ord('z') + 1)]
alpha_lis.remove("j")
for ch in cipher:
    if ch == 'j':
        ch = 'i'
    if ch not in cipher_list:
        cipher_list.append(ch)
for j in alpha_lis:
    if j not in cipher_list:
        cipher_list.append(j)
n = int(input().strip())
for _ in range(n):
    word_lis = deque(input().strip())
    group_lis = []
    while word_lis:
        i = word_lis.popleft()
        if i == 'j':
            i = 'i'
        if word_lis:
            j = word_lis.popleft()
            if j == 'j':
                j = 'i'
            if i == j:
                if i != 'x':
                    group_lis.append(i + 'x')
                    word_lis.appendleft(j)
                else:
                    group_lis.append(i + 'q')
                    word_lis.appendleft(j)
            else:
                group_lis.append(i + j)
        else:
            if i != 'x':
                group_lis.append(i + 'x')
            else:
                group_lis.append(i + 'q')
    output = ""
    for group in group_lis:
        first_index = cipher_list.index(group[0])
        second_index = cipher_list.index(group[1])
        row1 = first_index // 5
        col1 = first_index % 5
        row2 = second_index // 5
        col2 = second_index % 5
        if row1 == row2:
            output += cipher_list[first_index + 1] if col1 != 4 else
cipher_list[first_index - 4]
            output += cipher_list[second_index + 1] if col2 != 4 else
cipher_list[second_index - 4]
        elif col1 == col2:
            output += cipher_list[first_index + 5] if row1 != 4 else
cipher_list[first_index - 20]
            output += cipher_list[second_index + 5] if row2 != 4 else
cipher_list[second_index - 20]
        else:
            output += cipher_list[row1 * 5 + col2]
            output += cipher_list[row2 * 5 + col1]
    print(output)
```

File: 2025_fall_python_assignment\assignmentD\month_test\test6.py

```
# give you a bunch of NPU cores with different battery usage levels,
# for one core, the battery usage level is represented by two integers a, b, in a
periodic task, which means it will firstly
# use a units of battery to perform one task, then it will use b units of battery
to perform the next task, and then it will repeat this process.
# now you have a total battery capacity of C units, you want to know the maximum
number of tasks you can perform using these cores without exceeding the battery
```

```
capacity.
# each core can be used multiple times, and you can switch between cores at any
time, including in the middle of a core's periodic task.
import sys

def max_tasks(cores, m):
    # cores: list of (x, y) pairs, m: total energy
    n = len(cores)
    xs = [x for x, y in cores]
    pair_costs = [x + y for x, y in cores]

    xs_sorted = sorted(xs)
    pref = [0]  # prefix sums: pref[k] = sum of k smallest x
    for v in xs_sorted:
        pref.append(pref[-1] + v)

    min_pair = min(pair_costs)
    min_task_cost = min(min(xs), min(y for x, y in cores))

    # upper bound for tasks: each task至少消耗 min_task_cost
    lo, hi = 0, m // min_task_cost
    ans = 0

    def feasible(T):
        # return True if can finish T tasks within energy m
        # S ranges from parity = T%2 to min(n, T), step 2
        maxS = min(n, T)
        parity = T & 1
        best = 10**30
        # enumerate S with same parity as T
        for S in range(parity, maxS + 1, 2):
            P = (T - S) // 2
            cost = P * min_pair + pref[S]
            if cost < best:
                best = cost
                # small pruning: if best already <= m we can return True
                if best <= m:
                    return True
        return best <= m

    # binary search maximum T
    while lo <= hi:
        mid = (lo + hi) // 2
        if feasible(mid):
            ans = mid
            lo = mid + 1
        else:
            hi = mid - 1
    return ans

if __name__ == "__main__":
    # 读入 (与题目输入格式相符)
    data = sys.stdin.read().strip().split()
    it = iter(data)
    try:
        n = int(next(it))
        m = int(next(it))
    except StopIteration:
        print(0)
        sys.exit(0)
    cores = []
    for _ in range(n):
        x = int(next(it)); y = int(next(it))
        cores.append((x, y))
    print(max_tasks(cores, m))
```

File: 2025_fall_python_assignment\assignmentD\selective_exercise\all_ways.py

```
def find_all_ways(maze, n, m):
    visited = [[False for _ in range(m)] for _ in range(n)]
    num_steps = set()
    def dfs(x, y, steps):
        nonlocal num_steps
        if x == n - 1 and y == m - 1:
            num_steps.add(steps)
            return
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] == 0 and not
visited[nx][ny]:
                visited[nx][ny] = True
                dfs(nx, ny, steps + 1)
                visited[nx][ny] = False
    visited[0][0] = True
    dfs(0, 0, 0)
    return num_steps

n, m, wanted_steps = map(int, input().strip().split())
maze = [list(map(int, input().strip().split())) for _ in range(n)]
out_set = find_all_ways(maze, n, m)
if  wanted_steps in out_set:
    print("Yes")
else:
    print("No")
```

File: 2025_fall_python_assignment\assignmentD\selective_exercise\flight_problem.py

```
# give you a matrix, the flight cost between i and j is matrix[i][j], from i to j
and j to i have the same cost
# find the minimum cost to visit all the cities and return to the starting city
# the starting city can be any city
# optimized dp or dfs solution
def flight_problem(matrix: list):
    n = len(matrix)
    visited = [False] * n
    min_cost = float('inf')

    def dfs(city: int, count: int, cost: int, start_city):
        nonlocal min_cost
        if count == n and matrix[city][0] > 0:
            min_cost = min(min_cost, cost + matrix[city][start_city])
            return
        for next_city in range(n):
            if not visited[next_city]:
                visited[next_city] = True
                dfs(next_city, count + 1, cost + matrix[city][next_city],
start_city)
                visited[next_city] = False

    for start_city in range(n):
        visited[start_city] = True
        dfs(start_city, 1, 0, start_city)
        visited[start_city] = False
    return min_cost

n = int(input().strip())
matrix = []
for _ in range(n):
    row = list(map(int, input().strip().split()))
    matrix.append(row)

print(flight_problem(matrix))
```

File: 2025_fall_python_assignment\assignmentD\selective_exercise\llm_accelerate.py

```
core_num, battery_life = map(int, input().strip().split())
x_list = []
x_y_sum = []
for _ in range(core_num):
    x, y = map(int, input().strip().split())
    x_list.append(x)
    x_y_sum.append(x + y)
```

```
x_list.sort()
using_x_y_sum = min(x_y_sum)
max_rounds = 0
for i in range(core_num):
    if x_list[i] >= using_x_y_sum:
        max_rounds = max(max_rounds, rounds)
        break
    current_life = battery_life
    rounds = 0
    for j in range(i):
        if current_life >= x_list[j]:
            current_life -= x_list[j]
            rounds += 1
        else:
            max_rounds = max(max_rounds, rounds)
            break
    if current_life > 0:
        rounds += current_life // using_x_y_sum * 2
        max_rounds = max(max_rounds, rounds)
print(max_rounds)
```

File: 2025_fall_python_assignment\assignmentD\selective_exercise\matrix_fast_power.py

```
def matrix_multiply(a, b, mod = 1000000007):
    n = len(a)
    result = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                result[i][j] = (result[i][j] + a[i][k] * b[k][j]) % mod
    return result

def identity_matrix(n):
    output = [[0] * n for _ in range(n)]
    for i in range(n):
        output[i][i] = 1
    return output

def matrix_fast_power(a, k):
    n = len(a)
    output = identity_matrix(n)
    while k:
        if k & 1:
            output = matrix_multiply(output, a)
        a = matrix_multiply(a, a)
        k >>= 1
    return output
```

```
base_vector = [[1], [1]]
a_base = [[1, 1], [1, 0]]
wanted_index = int(input().strip())
if wanted_index <= 2:
    print(1)
    exit(0)
result_matrix = matrix_fast_power(a_base, wanted_index - 2)
result = (result_matrix[0][0] * base_vector[0][0] + result_matrix[0][1] *
base_vector[1][0]) % 1000000007
print(result)
```

File: 2025_fall_python_assignment\assignmentD\selective_exercise\place_apple.py

```
# give you m same apples and n same plates, find the sum of ways to put apples
into plates
# the plate can be empty
# add explaination
# 1,3,1 and 3,1,1 are the same so we don't count them twice
# dp solution
# test case input:
# 7 3
# output:
# 8
def place_apple(m: int, n: int) -> int:
    # dp[i][j] will be the number of ways to place i apples into j plates.
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Base case: There is one way to place any number of apples on one plate.
    for i in range(m + 1):
        dp[i][1] = 1

    # Base case: There is one way to place 0 apples on any number of plates.
    for j in range(n + 1):
        dp[0][j] = 1

    for i in range(1, m + 1):
        for j in range(2, n + 1):
            if i >= j:
                dp[i][j] = dp[i][j - 1] + dp[i - j][j]
            else:
                # Not enough apples to place one on each plate,
                # so at least one plate must be empty.
                dp[i][j] = dp[i][j - 1]

    return dp[m][n]

t = int(input().strip())
for _ in range(t):
    m, n = map(int, input().strip().split())
    print(place_apple(m, n))
```

File: 2025_fall_python_assignment\assignmentD\selective_exercise\weighted_matrix.py

```
def find_the_max_weight(maze, n, m):
    max_weight = float('-inf')
    visited = [[False for _ in range(m)] for _ in range(n)]
    def dfs(x, y, current_weight, step_list=[]):
        nonlocal max_weight
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        if x == n - 1 and y == m - 1:
            if current_weight > max_weight:
                max_weight = current_weight

            return
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny]:
                visited[nx][ny] = True
                dfs(nx, ny, current_weight + maze[nx][ny])
                visited[nx][ny] = False
    visited[0][0] = True
    dfs(0, 0, maze[0][0])
    return max_weight

m, n = map(int, input().strip().split())
maze = [list(map(int, input().strip().split())) for _ in range(m)]
print(find_the_max_weight(maze, m, n))
```

File: 2025_fall_python_assignment\dawn_before_final\basketball_exercise.py

```
def choose_the_best(lis1, lis2, n) -> int:
    dp = [[0] * (n + 1) for _ in range(3)]

    for i in range(1, n + 1):
        dp[0][i] = max(dp[2][i - 1] + lis1[i - 1], dp[1][i - 1] + lis1[i - 1])
        dp[1][i] = max(dp[2][i - 1] + lis2[i - 1], dp[0][i - 1] + lis2[i - 1])
        dp[2][i] = max(dp[1][i - 1], dp[0][i - 1])

    return max(dp[0][n], dp[1][n])

n = int(input().strip())
```

```
lis1 = list(map(int, input().strip().split()))
lis2 = list(map(int, input().strip().split()))
print(choose_the_best(lis1, lis2, n))
```

File: 2025_fall_python_assignment\dawn_before_final\binary_count.py

```
# calculate how many binary numbers in 1 to n has k 1s in their binary
representation
# large n up to 10^9
def binary_count(n: int, k: int) -> int:
    # Precompute binomial coefficients C(n, k) for n, k <= 30
    C = [[0] * 31 for _ in range(31)]
    for i in range(31):
        C[i][0] = 1
        for j in range(1, i + 1):
            C[i][j] = C[i - 1][j - 1] + C[i - 1][j]

    count = 0
    ones_count = 0
    for i in range(30, -1, -1):
        if n & (1 << i):
            # If the i-th bit is set, add combinations of remaining bits
            if k - ones_count >= 0:
                count += C[i][k - ones_count]
            ones_count += 1
            if ones_count > k:
                break
        # Check if the number itself has exactly k ones
    if ones_count == k:
        count += 1

    return count


n, k = map(int, input().strip().split())
print(binary_count(n, k))
```

File: 2025_fall_python_assignment\dawn_before_final\boredom.py

```
def max_points(lis) -> int:
    from collections import Counter
    count = Counter(lis)
    dp_table = [0] * (max(lis) + 1)
    dp_table[1] = count[1] * 1
    for i in range(2, len(dp_table)):
        dp_table[i] = max(dp_table[i - 1], dp_table[i - 2] + count[i] * i)
    return dp_table[-1]
n = int(input().strip())
lis = list(map(int, input().strip().split()))
print(max_points(lis))
```

File: 2025_fall_python_assignment\dawn_before_final\cat_coin.py

```
def cat_coin(coins):
    coins.sort(reverse = True)
    odd_coins = [coin for coin in coins if coin % 2 == 1]
    even_coins = [coin for coin in coins if coin % 2 == 0]
    result = [0] * len(coins)

    if not odd_coins:
        return result
    result[0] = odd_coins[0]

    if not even_coins:
        for i in range(len(coins)):
            if i % 2 == 0:
                result[i] = odd_coins[0]
            else:
                result[i] = 0
        return result

    for i in range(1, len(even_coins) + 1):
        result[i] = result[i - 1] + even_coins[i-1]

    start_point = len(even_coins) + 1

    if len(odd_coins) <= 2:
        for i in range(start_point, len(coins)):
            result[i] = 0
    else:
        ans1 = result[start_point - 1]
        ans2 = result[start_point - 1] - even_coins[-1]
        for i in range(start_point, len(coins)):
            if (i - start_point) % 2 == 0:
                if i == len(coins) - 1:
                    result[i] = 0
                    continue
                result[i] = ans2
            else:
                result[i] = ans1
    return result

t = int(input().strip())
for _ in range(t):
    n = int(input().strip())
    coins = list(map(int, input().strip().split()))
    res = cat_coin(coins)
    print(' '.join(map(str, res)))
```

File: 2025_fall_python_assignment\dawn_before_final\chinese_chess.py

```
from collections import deque

start_pos = tuple(map(int, input().strip().split()))
end_pos = tuple(map(int, input().strip().split()))
pieces = int(input().strip())
pieces_list = set()
for _ in range(pieces):
    piece_pos = tuple(map(int, input().strip().split()))
    pieces_list.add(piece_pos)

queue = deque()
parent_dic = dict()
visited = set()
queue.append(start_pos)
parent_dic[start_pos] = None
visited.add(start_pos)
directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2,
1)]

while queue:
    current_pos = queue.popleft()
    visited.add(current_pos)
    if current_pos == end_pos:
        break
    for dx, dy in directions:
        next_x = current_pos[0] + dx
        next_y = current_pos[1] + dy
        if tuple([next_x - dx//abs(dx), next_y - dy//abs(dy)]) in pieces_list:
            continue
        next_pos = (next_x, next_y)
        if 0 <= next_x <= 10 and 0 <= next_y <= 10 and next_pos not in visited and
next_pos not in pieces_list:
            parent_dic[next_pos] = current_pos
            queue.append(next_pos)

count = 1
while queue:
    current_pos = queue.popleft()
    if current_pos == end_pos:
        count += 1

if count > 1:
    print(count)
```

```
else:
    path = []
    backtrack_pos = end_pos
    while backtrack_pos is not None:
        path.append(backtrack_pos)
        backtrack_pos = parent_dic[backtrack_pos]
    path.reverse()
    output_str = '-'.join([f"({pos[0]},{pos[1]})" for pos in path])
    print(output_str)
```

File: 2025_fall_python_assignment\dawn_before_final\cows.py

```
# give you n stalls and c cows, find the largest minimum distance between cows if you put them in the stalls
# return the minimum distance
import sys

def can_place_cows(stalls: list, c: int, min_dist: int) -> bool:
    count = 1  # Place the first cow in the first stall
    last_position = stalls[0]

    for i in range(1, len(stalls)):
        if stalls[i] - last_position >= min_dist:
            count += 1
            last_position = stalls[i]
            if count == c:
                return True

    return False

def cows(stalls: list, c: int) -> int:
    stalls.sort()
    left, right = 0, stalls[-1] - stalls[0]
    best_dist = 0

    while left <= right:
        mid = (left + right) // 2
        if can_place_cows(stalls, c, mid):
            best_dist = mid
            left = mid + 1
        else:
            right = mid - 1

    return best_dist

n, c = map(int, input().strip().split())
lines = sys.stdin.readlines()
stalls = [int(line.strip()) for line in lines]
print(cows(stalls, c))
```

File: 2025_fall_python_assignment\dawn_before_final\distinct_mushroom.py

```python
# give you a list of mushrooms, count the ways to select a part of the list so mushrooms in that part have less than k colors or k colors
# one number represents one color
# sliding window needed
n, k = map(int, input().split())
mushrooms = list(map(int, input().split()))
left = 0
color_count = {}
result = 0
for right in range(n):
    color = mushrooms[right]
    if color in color_count:
        color_count[color] += 1
    else:
        color_count[color] = 1
    while len(color_count) > k:
        left_color = mushrooms[left]
        color_count[left_color] -= 1
        if color_count[left_color] == 0:
            del color_count[left_color]
        left += 1
    result += right - left + 1
print(result)
```

File: 2025_fall_python_assignment\dawn_before_final\divide_islands.py

```python
# give you an n*m grid representing a map where . means water and X means land.
# An island is a maximal 4-directionally connected group of land cells.
# there are 3 lands, find the minimum number of water cells you must convert to land to connect all lands into one island.
import sys
from collections import deque
def min_water_to_connect_islands(grid):
    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    def bfs_mark_island(r, c, island_id):
        queue = deque([(r, c)])
        grid[r][c] = island_id
        while queue:
            x, y = queue.popleft()
            for dr, dc in directions:
                nr, nc = x + dr, y + dc
                if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 'X':
                    grid[nr][nc] = island_id
                    queue.append((nr, nc))

    island_id = 2
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 'X':
                bfs_mark_island(r, c, island_id)
                island_id += 1

    # Helper function to calculate distance from a specific island to all other
cells
    def bfs_distance_from_island(island_id):
        dist = [[float('inf')] * cols for _ in range(rows)]
        queue = deque()

        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == island_id:
                    dist[r][c] = 0
                    queue.append((r, c))

        while queue:
            r, c = queue.popleft()
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols:
                    if dist[nr][nc] == float('inf'):
                        dist[nr][nc] = dist[r][c] + 1
                        queue.append((nr, nc))
        return dist

    # Calculate distances from each of the 3 islands
    dist_from_island_2 = bfs_distance_from_island(2)
    dist_from_island_3 = bfs_distance_from_island(3)
    dist_from_island_4 = bfs_distance_from_island(4)

    min_water = float('inf')

    # 1. Strategy A: Find optimal meeting point (Y-shape or single center)
    for r in range(rows):
        for c in range(cols):
            d2 = dist_from_island_2[r][c]
            d3 = dist_from_island_3[r][c]
            d4 = dist_from_island_4[r][c]

            if d2 != float('inf') and d3 != float('inf') and d4 != float('inf'):
                min_water = min(min_water, d2 + d3 + d4 - 2)

    # 2. Strategy B: Sequential connections (Linear chain)
    # Calculate minimum distance between pairs of islands
    def get_min_dist(dist_grid, target_island_id):
        min_d = float('inf')
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == target_island_id:
                    min_d = min(min_d, dist_grid[r][c])
        return max(0, min_d - 1) # Distance is steps - 1 water cells

    dist_2_3 = get_min_dist(dist_from_island_2, 3)
    dist_2_4 = get_min_dist(dist_from_island_2, 4)
    dist_3_4 = get_min_dist(dist_from_island_3, 4)

    # Check all linear combinations
    min_water = min(min_water, dist_2_3 + dist_3_4) # 2-3-4
    min_water = min(min_water, dist_2_4 + dist_3_4) # 2-4-3
    min_water = min(min_water, dist_2_3 + dist_2_4) # 3-2-4

    return min_water

n, m = map(int, input().split())
grid = [list(input().strip()) for _ in range(n)]
print(min_water_to_connect_islands(grid))
```

File: 2025_fall_python_assignment\dawn_before_final\exempt_from_homework.py

```
n = int(input())
arr = list(map(int, input().split()))
ways = 0
if arr[0] != 0:
    ways += 1
for i in range(1, n):
    if arr[i-1] < i and arr[i] > i:
        ways += 1
if arr[-1] != n:
    ways += 1

print(ways)
```

File: 2025_fall_python_assignment\dawn_before_final\expeditions.py

```
import heapq

num_of_fuel_stations = int(input().strip())
fuel_stations_inf = []
```

```
for _ in range(num_of_fuel_stations):
    position, fuel = map(int, input().strip().split())
    fuel_stations_inf.append((position, -fuel))

fuel_stations_inf.sort(reverse = True)
distance, total_fuel = map(int, input().strip().split())
pointer = 0
reachable_stations = []
count = 0
while True:
    while pointer < num_of_fuel_stations and distance - fuel_stations_inf[pointer]
[0] <= total_fuel:
        heapq.heappush(reachable_stations, fuel_stations_inf[pointer][1])
        pointer += 1
    if total_fuel < distance:
        if not reachable_stations:
            count = -1
            break
        total_fuel += -heapq.heappop(reachable_stations)
        count += 1
    else:
        break
print(count)
```

File: 2025_fall_python_assignment\dawn_before_final\find_biggest_XOR.py

```
class Solution:
    def findMaximumXOR(self, nums: List[int]) -> int:
        root =  [None, None]

        for num in nums:
            node = root
            for i in range(31, -1, -1):
                bit = (num >> i) & 1
                if not node[bit]:
                    node[bit] = [None, None]
                node = node[bit]

        max_xor = 0
        for num in nums:
            node = root
            curr_xor = 0
            for i in range(31, -1, -1):
                bit = (num >> i) & 1
                toggled_bit = 1 - bit
                if node[toggled_bit]:
                    curr_xor |= (1 << i)
                    node = node[toggled_bit]
                else:
                    node = node[bit]
                max_xor = max(max_xor, curr_xor)

        return max_xor
```

File: 2025_fall_python_assignment\dawn_before_final\find_the_multiple.py

```
# given a positive integer n, find a nonzero multiple m of n whose decimal
representation consists of only digit 0 and 1
# add explaination
# use bfs to find the smallest such multiple
def find_multiple(n: int) -> str:
    from collections import deque

    queue = deque()
    visited = set()

    queue.append('1')
    while queue:
        current = queue.popleft()
        current_mod = int(current) % n

        if current_mod == 0:
            return current

        if current_mod not in visited:
            visited.add(current_mod)
            queue.append(current + '0')
            queue.append(current + '1')

    return ""

while True:
    n = int(input().strip())
    if n == 0:
        break
    print(find_multiple(n))
```

File: 2025_fall_python_assignment\dawn_before_final\flowers_again.py

```
# you can eat white or red flowers for dinner, however you can only eat white
flowers in size k(consecutively)
# determine how many ways you can eat from a to b, a, b means the total number of
flowers you will eat
# use dp, prefix sum and deque to achieve an rolling window
# the output may be large, so use modulo 10^9 + 7
```

```python
# n indicates the total number of testcases
from collections import deque
MOD = 10**9 + 7
n, k = map(int, input().strip().split())
largest_end = -1
cases = []

for _ in range(n):
    start, end = map(int, input().strip().split())
    largest_end = max(largest_end, end)
    cases.append((start, end))

prefix_sum = [0] * (largest_end + 1)
dp_window = deque(maxlen = k + 1)
dp_window.append(1)
for i in range(1, largest_end + 1):
    if i < k:
        dp_value = dp_window[-1] % MOD
        prefix_sum[i] = prefix_sum[i-1] + dp_value
        dp_window.append(dp_value)
    elif i == k:
        dp_value = (dp_window[-1] + 1) % MOD
        prefix_sum[i] = prefix_sum[i-1] + dp_value
        dp_window.append(dp_value)
    else:
        dp_value = dp_window[-1] + dp_window[1]
        dp_value %= MOD
        prefix_sum[i] = prefix_sum[i-1] + dp_value
        dp_window.append(dp_value)

for start, end in cases:
    result = (prefix_sum[end] - prefix_sum[start - 1]) % MOD if start > 1 else
prefix_sum[end] % MOD
    print(result)
```

File: 2025_fall_python_assignment\dawn_before_final\hide_the_needle.py

```python
def hide_string(s: str, hide: str) -> str:
    s_lis = list(s)
    hide_lis = list(hide)
    pointer = 0
    for char in s_lis:
        try:
            hide_lis.remove(char)
        except ValueError:
            return "Impossible"

    hide_lis.sort()
    ans = []
    for char in s_lis:
        for i in range(len(hide_lis)):
            if hide_lis[i] < char:
                ans.append(hide_lis[i])
                if i == len(hide_lis) - 1:
                    hide_lis = []
            else:
                hide_lis = hide_lis[i:]
                break
        ans.append(char)
    ans.extend(hide_lis)
    return ''.join(ans)

t = int(input().strip())
for _ in range(t):
    s = input().strip()
    hide = input().strip()
    result = hide_string(s, hide)
    print(result)
```

File: 2025_fall_python_assignment\dawn_before_final\implement_trie.py

```
class Trie:

    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

    def insert(self, word: str) -> None:
        for char in word:
            if char not in self.children:
                self.children[char] = Trie()
            self = self.children[char]
        self.is_end_of_word = True

    def search(self, word: str) -> bool:
        for char in word:
            if char not in self.children:
                return False
            self = self.children[char]
        return self.is_end_of_word

    def startsWith(self, prefix: str) -> bool:
        for char in prefix:
            if char not in self.children:
                return False
            self = self.children[char]
        return True
```

```
# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

File: 2025_fall_python_assignment\dawn_before_final\interesting_drink.py

```
from functools import lru_cache

@lru_cache(None)
def interesting_drink(m_values) -> int:
        # Binary search to find the number of shops with price <= m
    left, right = 0, len(prices)
    while left < right:
        mid = (left + right) // 2
        if prices[mid] <= m:
            left = mid + 1
        else:
            right = mid
    result = left
    return result

n = int(input().strip())
prices = list(map(int, input().strip().split()))
prices.sort()
q = int(input().strip())
for _ in range(q):
    m = int(input().strip())
    print(interesting_drink(m))
```

File: 2025_fall_python_assignment\dawn_before_final\least_ball_num.py

```
class Solution:
    def minimumSize(self, nums: List[int], maxOperations: int) -> int:
        def can_divide(size: int) -> bool:
            operations = 0
            for num in nums:
                if num > size:
                    operations += (num - 1) // size
                if operations > maxOperations:
                    return False
            return True

        left, right = 1, max(nums)
        while left < right:
            mid = (left + right) // 2
            if can_divide(mid):
                right = mid
            else:
                left = mid + 1
        return left
```

File: 2025_fall_python_assignment\dawn_before_final\longest_words.py

```
class Solution:
    def longestWord(self, words: List[str]) -> str:
        class TrieNode:
            def __init__(self):
                self.children = {}
                self.is_end = False

        class Trie:
            def __init__(self):
                self.root = TrieNode()

            def insert(self, word: str) -> None:
                node = self.root
                for char in word:
                    if char not in node.children:
                        node.children[char] = TrieNode()
                    node = node.children[char]
                node.is_end = True

        solution_length = float('-inf')
        solution_bucket = []
        trie = Trie()
        for word in words:
            trie.insert(word)

        words.sort(key = lambda x: len(x), reverse = True)
        for word in words:
            if len(word) < solution_length:
                break
            node = trie.root
            for i, char in enumerate(word):
                if i == len(word) - 1:
                    solution_bucket.append(word)
                    solution_length = len(word)
                if char in node.children:
                    node = node.children[char]
                    if not node.is_end:
                        break
        solution_bucket.sort()
        return solution_bucket[0] if solution_bucket else ""
```

File: 2025_fall_python_assignment\dawn_before_final\magic_dictionary.py

```python
class MagicDictionary:

    def __init__(self):
        self.children = {}
        self.is_end = False

    def buildDict(self, dictionary: List[str]) -> None:
        for word in dictionary:
            node = self
            for char in word:
                if char not in node.children:
                    node.children[char] = MagicDictionary()
                node = node.children[char]
            node.is_end = True


    def search(self, searchWord: str) -> bool:
        def _dfs(self, node, index: int, modified: bool) -> bool:
            if index == len(searchWord):
                return modified and node.is_end
            char = searchWord[index]
            if char in node.children:
                if _dfs(self, node.children[char], index + 1, modified):
                    return True
            if not modified:
                for child_char, child_node in node.children.items():
                    if child_char != char:
                        if _dfs(self, child_node, index + 1, True):
                            return True

            return False
        return _dfs(self, self, 0, False)

from typing import List
# Your MagicDictionary object will be instantiated and called as such:
# obj = MagicDictionary()
# obj.buildDict(dictionary)
# param_2 = obj.search(searchWord)
```

File: 2025_fall_python_assignment\dawn_before_final\map_sum.py

```python
class MapSum:

    def __init__(self):
        self.children = {}
        self.value = 0
    def insert(self, key: str, val: int) -> None:
        node = self
        for char in key:
            if char not in node.children:
                node.children[char] = MapSum()
            node = node.children[char]
        node.value = val


    def sum(self, prefix: str) -> int:
        node = self
        for char in prefix:
            if char not in node.children:
                return 0
            node = node.children[char]

        def dfs(n: MapSum) -> int:
            total = n.value
            for child in n.children:
                total += dfs(n.children[child])
            return total

        return dfs(node)



# Your MapSum object will be instantiated and called as such:
# obj = MapSum()
# obj.insert(key,val)
# param_2 = obj.sum(prefix)
```

File: 2025_fall_python_assignment\dawn_before_final\matrix.py

```
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        if not mat or not mat[0]:
            return mat

        rows, cols = len(mat), len(mat[0])
        dist = [[float('inf')] * cols for _ in range(rows)]

        # First pass: check for top and left
        for r in range(rows):
            for c in range(cols):
                if mat[r][c] == 0:
                    dist[r][c] = 0
                else:
                    if r > 0:
                        dist[r][c] = min(dist[r][c], dist[r - 1][c] + 1)
                    if c > 0:
                        dist[r][c] = min(dist[r][c], dist[r][c - 1] + 1)

        # Second pass: check for bottom and right
        for r in range(rows - 1, -1, -1):
            for c in range(cols - 1, -1, -1):
                if r < rows - 1:
                    dist[r][c] = min(dist[r][c], dist[r + 1][c] + 1)
                if c < cols - 1:
                    dist[r][c] = min(dist[r][c], dist[r][c + 1] + 1)

        return dist
```

File: 2025_fall_python_assignment\dawn_before_final\min_mutation.py

```
class Solution:
    def minMutation(self, startGene: str, endGene: str, bank: List[str]) -> int:
        from collections import deque
        bank_set = set(bank)
        if endGene not in bank_set:
            return -1
        gene_chars = ['A', 'C', 'G', 'T']
        queue = deque([(startGene, 0)])
        visited = set([startGene])

        while queue:
            current_gene, steps = queue.popleft()
            if current_gene == endGene:
                return steps
            for i in range(len(current_gene)):
                for char in gene_chars:
                    if char != current_gene[i]:
                        mutated_gene = current_gene[:i] + char +
current_gene[i+1:]
                        if mutated_gene in bank_set and mutated_gene not in
visited:
                            visited.add(mutated_gene)
                            queue.append((mutated_gene, steps + 1))
        return -1
```

File: 2025_fall_python_assignment\dawn_before_final\monthly_spend.py

```
# give you n integers in a list, devide them into m parts, such that the maximum
sum of each part is minimized
# parts must be continuous
def divide_into_parts(nums, m):
    def can_divide(max_sum):
        current_sum = 0
        parts = 1
        for num in nums:
            current_sum += num
            if current_sum > max_sum:
                parts += 1
                current_sum = num
                if parts > m:
                    return False
        return True

    left, right = max(nums), sum(nums)
    while left < right:
        mid = (left + right) // 2
        if can_divide(mid):
            right = mid
        else:
            left = mid + 1
    return left

n, m = map(int, input().split())
nums = []
for _ in range(n):
    num = int(input())
    nums.append(num)
print(divide_into_parts(nums, m))
```

File: 2025_fall_python_assignment\dawn_before_final\phone_number.py

```
# give you a bunch of phone numbers, decide if they are consistent
# consistent means no number is prefix of another number
# use trie to solve this problem
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_number = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, number: str) -> bool:
        current = self.root
        for digit in number:
            if digit not in current.children:
                current.children[digit] = TrieNode()
            current = current.children[digit]
            if current.is_end_of_number:
                return False  # Found a prefix
        if current.children:
                return False  # Current number is a prefix of another number
            current.is_end_of_number = True
            return True

t = int(input().strip())
for _ in range(t):
    n = int(input().strip())
    trie = Trie()
    consistent = True
    for _ in range(n):
        number = input().strip()
        if not trie.insert(number):
            consistent = False
    print("YES" if consistent else "NO")
```

File: 2025_fall_python_assignment\dawn_before_final\pile_pigs.py

```
import sys
# need to read multiple lines in once and store each line into a list
input_lines = sys.stdin.read().strip().split('\n')
stack = []
min_stack = []

for line in input_lines:
    if len(line) != 3:
        a, b = map(str, line.strip().split())
        b = int(b)
        stack.append(b)
        if not min_stack or b <= min_stack[-1]:
            min_stack.append(b)
    elif line == 'pop':
        if stack:
            top = stack.pop()
            if top == min_stack[-1]:
                min_stack.pop()
    elif line == 'min':
        if stack:
            print(min_stack[-1])
```

File: 2025_fall_python_assignment\dawn_before_final\pku_game.py

```
n, k = map(int, input().strip().split())
player_dict = dict()
total_list = []
for i in range(n):
    temp_list = list(map(int, input().strip().split()))
    total_list.extend(temp_list)
```

```
        player_dict[i] = set(temp_list)

    losing_time = [0] * n
    length = n * k
    for i in total_list:
        for j in range(n-1, -1, -1):
            if i in player_dict[j]:
                losing_time[j] += 1
                break

    for player in losing_time:
        print ('%.9f' % (player / length))
```

File: 2025_fall_python_assignment\dawn_before_final\receive_rain.py

```
# give you n integers > 0 representing the height of one pillar in a row.
# When it rains, water is trapped between the pillars.
# calculate how much water is trapped after raining.
# The width of each pillar is 1.
def trap_rain_water(heights):
    if not heights:
        return 0

    n = len(heights)
    left_max = [0] * n
    right_max = [0] * n

    left_max[0] = heights[0]
    for i in range(1, n):
        left_max[i] = max(left_max[i - 1], heights[i])

    right_max[n - 1] = heights[n - 1]
    for i in range(n - 2, -1, -1):
        right_max[i] = max(right_max[i + 1], heights[i])

    trapped_water = 0
    for i in range(n):
        trapped_water += min(left_max[i], right_max[i]) - heights[i]

    return trapped_water

n = int(input().strip())
heights = list(map(int, input().strip().split()))
print(trap_rain_water(heights))
```

File: 2025_fall_python_assignment\dawn_before_final\replace_words.py

```
class Solution:
    def replaceWords(self, dictionary: List[str], sentence: str) -> str:
        class TrieNode:
            def __init__(self):
                self.children = {}
                self.is_end_of_word = False

        class Trie:
            def __init__(self):
                self.root = TrieNode()

            def insert(self, word: str) -> None:
                node = self.root
                for char in word:
                    if char not in node.children:
                        node.children[char] = TrieNode()
                    node = node.children[char]
                node.is_end_of_word = True

        ans_bucket = []
        trie = Trie()
        for word in dictionary:
            trie.insert(word)

        for word in sentence.split():
            node = trie.root
            prefix = ""
            notfind = False
            for i, char in enumerate(word):
                if char in node.children:
                    prefix += char
                    node = node.children[char]
                    if node.is_end_of_word:
                        break
                else:
                    notfind = True
                    break

            if not notfind:
                ans_bucket.append(prefix)
            else:
                ans_bucket.append(word)

        return " ".join(ans_bucket)
```

File: 2025_fall_python_assignment\dawn_before_final\search_words.py

```
class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        class TrieNode:
            def __init__(self):
                self.children = {}
                self.is_end_of_word = False

        class Trie:
            def __init__(self):
                self.root = TrieNode()

            def insert(self, word: str) -> None:
                node = self.root
                for char in word:
                    if char not in node.children:
                        node.children[char] = TrieNode()
                    node = node.children[char]
                node.is_end_of_word = True

        trie = Trie()
        for word in words:
            trie.insert(word)

        def dfs(x: int, y: int, node: TrieNode, path: str) -> None:
            if node.is_end_of_word:
                result.add(path)
            node.is_end_of_word = False  # Avoid duplicate entries

            if x < 0 or x >= len(board) or y < 0 or y >= len(board[0]):
                return

            char = board[x][y]
            if char not in node.children:
                return

            board[x][y] = '#'  # Mark as visited
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                dfs(x + dx, y + dy, node.children[char], path + char)
            board[x][y] = char  # Restore original character

        result = set()
        for i in range(len(board)):
            for j in range(len(board[0])):
                dfs(i, j, trie.root, "")
        return list(result)
```

File: 2025_fall_python_assignment\dawn_before_final\search_words_basic.py

```python
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def backtrack(x, y, index):
            if index == len(word):
                return True
            if x < 0 or x >= len(board) or y < 0 or y >= len(board[0]) or board[x]
[y] != word[index]:
                return False
            char = board[x][y]
            board[x][y] = '#'
            found = (backtrack(x + 1, y, index + 1) or
                     backtrack(x - 1, y, index + 1) or
                     backtrack(x, y + 1, index + 1) or
                     backtrack(x, y - 1, index + 1))
            board[x][y] = char
            return found

        for i in range(len(board)):
            for j in range(len(board[0])):
                if backtrack(i, j, 0):
                    return True

        return False
```

File: 2025_fall_python_assignment\dawn_before_final\skiing.py

```python
from functools import lru_cache

r, c = map(int, input().strip().split())
grid = [list(map(int, input().strip().split())) for _ in range(r)]
dp = [[1] * c for _ in range(r)]
visited = [[False] * c for _ in range(r)]
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
@lru_cache(None)
def dfs(x: int, y: int):
    if visited[x][y]:
        return dp[x][y]
    visited[x][y] = True
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < r and 0 <= ny < c and grid[nx][ny] < grid[x][y]:
            dp[x][y] = max(dp[x][y], dfs(nx, ny) + 1)
    return dp[x][y]
max_path = 0
for i in range(r):
    for j in range(c):
        max_path = max(max_path, dfs(i, j))
print(max_path)
```

File: 2025_fall_python_assignment\dawn_before_final\snake_ladder.py

```python
# play snakes and ladders game, give you a board, find the minimum number of moves
to reach the end
# the board is n x n, with some cells having snakes or ladders
class Solution:
    def snakesAndLadders(self, board: List[List[int]]) -> int:
        from collections import deque

        n = len(board)
        def get_coordinates(s):
            quot, rem = divmod(s - 1, n)
            row = n - 1 - quot
            col = rem if quot % 2 == 0 else n - 1 - rem
            return row, col

        visited = set()
        queue = deque([(1, 0)])  # (square number, moves)
        visited.add(1)
        while queue:
            position, moves = queue.popleft()
            if position == n * n:
                return moves
            for i in range(1, 7):
                next_pos = position + i
                if next_pos > n * n:
                    continue
                r, c = get_coordinates(next_pos)
                if board[r][c] != -1:
                    next_pos = board[r][c]
                if next_pos not in visited:
                    visited.add(next_pos)
                    queue.append((next_pos, moves + 1))
        return -1
```

File: 2025_fall_python_assignment\dawn_before_final\swapstring.py

```python
class Solution:
    def smallestStringWithSwaps(self, s: str, pairs: List[List[int]]) -> str:
        parent = list(range(len(s)))

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])
            return parent[x]

        def union(x, y):
            rootX = find(x)
            rootY = find(y)
            if rootX != rootY:
                parent[rootY] = rootX

        for x, y in pairs:
            union(x, y)

        from collections import defaultdict
        components = defaultdict(list)
        for i in range(len(s)):
            root = find(i)
            components[root].append(i)

        res = list(s)
        for indices in components.values():
            chars = [s[i] for i in indices]
            chars.sort()
            indices.sort()
            for i, char in zip(indices, chars):
                res[i] = char

        return ''.join(res)
```

File: 2025_fall_python_assignment\dawn_before_final\the_reachable_building.py

```
class Solution:
    def furthestBuilding(self, heights: List[int], bricks: int, ladders: int) ->
int:
        import heapq

        min_heap = []
        n = len(heights)

        for i in range(n - 1):
            diff = heights[i + 1] - heights[i]
            if diff > 0:
                heapq.heappush(min_heap, diff)
            if len(min_heap) > ladders:
                bricks -= heapq.heappop(min_heap)
            if bricks < 0:
                return i

        return n - 1
```

File: 2025_fall_python_assignment\dawn_before_final\vacations.py

```
def busy_vacation(days, states):
```

File: 2025_fall_python_assignment\dawn_before_final\walk_hill.py

```
# m, n, p means number of rows, columns and number of test cases
# then give you a matrix of n*m representing a hilly area, # means unpassable, number means height
# next p lines give you start and end point like x1 y1 x2 y2
# find the minimum effort to walk from start to end
# effort is defined as the absolute height difference between two consecutive cells in the path
# dijkstra solution
import heapq
def min_effort_path(grid, start, end):
    rows, cols = len(grid), len(grid[0])
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    effort = [[float('inf')] * cols for _ in range(rows)]
    effort[start[0]][start[1]] = 0
    min_heap = [(0, start[0], start[1])]
    if grid[start[0]][start[1]] == '#' or grid[end[0]][end[1]] == '#':
        return "NO"

    while min_heap:
        current_effort, x, y = heapq.heappop(min_heap)
        if (x, y) == end:
            return current_effort
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] != '#':
                height_diff = abs(int(grid[nx][ny]) - int(grid[x][y]))
                next_effort = max(current_effort, height_diff)
                if next_effort < effort[nx][ny]:
                    effort[nx][ny] = next_effort
                    heapq.heappush(min_heap, (next_effort, nx, ny))
    return "NO"

m, n, p = map(int, input().strip().split())
grid = [list(input().split()) for _ in range(m)]
for _ in range(p):
    x1, y1, x2, y2 = map(int, input().strip().split())
    start = (x1, y1)
    end = (x2, y2)
    result = min_effort_path(grid, start, end)
    print(result)
```

File: 2025_fall_python_assignment\dawn_before_final\binary_search_practice\find_the_element.py

```
n, target = map(int, input().split())
arr = list(map(int, input().split()))
def binary_search(arr, target):
    left, right = 0, n - 1
    while left < right:
        mid = (left + right) // 2
        if arr[mid] <= target:
            right = mid
        else:
            left = mid + 1
    return left if arr[left] == target else -1

result = binary_search(arr, target)
print(result)
```

File: 2025_fall_python_assignment\dawn_before_final\binary_search_practice\find_the_element_plus.py

```
n, target = map(int, input().split())
arr = list(map(int, input().split()))
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
```

File: 2025_fall_python_assignment\dawn_before_final\dijkstra_algorism\new_small_game.py

```
def judge_and_calculate_the_shortest_path(w, h, board, start, goal):
    from collections import deque
    dist = [[float('inf')] * (w + 2) for _ in range(h + 2)]
    queue = deque([(start[0], start[1])])
    dist[start[1]][start[0]] = 0

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    while queue:
        cx, cy = queue.popleft()
        for dx, dy in directions:
            nx, ny = cx + dx, cy + dy

            # 射线逻辑：沿一个方向一直走
            while 0 <= nx <= w + 1 and 0 <= ny <= h + 1:
                # 如果撞到了卡片
                if board[ny][nx] == 'X':
                    # 特判：如果是终点，更新距离但结束此射线
                    if nx == goal[0] and ny == goal[1]:
                        if dist[ny][nx] > dist[cy][cx] + 1:
                            dist[ny][nx] = dist[cy][cx] + 1
                            # 终点不需要入队去扩展，因为它不能穿过
                    break

                # 如果是空格，尝试更新最短线段数
                if dist[ny][nx] > dist[cy][cx] + 1:
                    dist[ny][nx] = dist[cy][cx] + 1
                    queue.append((nx, ny))

                nx += dx
                ny += dy

    return dist[goal[1]][goal[0]] if dist[goal[1]][goal[0]] != float('inf') else
-1
board_num = 1
while True:
    w, h = map(int, input().split())
    if w == 0 and h == 0:
        break
    board = [[' '] * (w + 2)] + [[' '] + list(input()) + [' '] for _ in range(h)]
+ [[' '] * (w + 2)]
    print(f'Board #{board_num}:')
    board_num += 1
    pairs = 1
    while True:
        start_x, start_y, goal_x, goal_y = map(int, input().split())
        if start_x == 0 and start_y == 0 and goal_x == 0 and goal_y == 0:
            break
        result = str(judge_and_calculate_the_shortest_path(w, h, board, (start_x,
start_y), (goal_x, goal_y)))
        if result == '-1':
            result = 'impossible.'
        else:
            result += ' segments.'
        print(f'Pair {pairs}: {result}')
        pairs += 1
    print()  # Blank line after each board
```

File: 2025_fall_python_assignment\dawn_before_final\dijkstra_algorism\small_

game.py

```python
# notice that in this file, w stands for the width of the board (number of
columns), and h stands for the height of the board (number of rows).
# meanwhile, start and goal are tuples in the form of (x, y), where x starts from
1 to w and y starts from 1 to h.

def judge_and_calculate_the_shortest_path(w, h, board, start, goal):
    from collections import deque
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    queue = deque()
    queue.append((start[0]-1, start[1]-1, 0))  # (x, y, distance)
    visited = set()
    visited.add((start[0]-1, start[1]-1))
    while queue:
        x, y, dist = queue.popleft()
        if (x, y) == (goal[0]-1, goal[1]-1):
            return dist
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < w and 0 <= ny < h:
                if board[ny][nx] != 'X' and (nx, ny) not in visited:
                    visited.add((nx, ny))
                    queue.append((nx, ny, dist + 1))
            else:
                if (nx, ny) not in visited:
                    visited.add((nx, ny))
                    queue.append((nx, ny, dist + 1))

    return -1

while True:
    board_num = 1
    w, h = map(int, input().split())
    if w == 0 and h == 0:
        break
    board = [list(input().strip()) for _ in range(h)]
    print(f'Board #{board_num}:')
    board_num += 1
    while True:
        pairs = 1
        start_x, start_y, goal_x, goal_y = map(int, input().split())
        if start_x == 0 and start_y == 0 and goal_x == 0 and goal_y == 0:
            break
        result = str(judge_and_calculate_the_shortest_path(w, h, board, (start_x,
start_y), (goal_x, goal_y)))
        if result == -1:
            result = 'impossible.'
        else:
            result += ' segments.'
        print(f'Pair {pairs}: {result}')
        pairs += 1
    print()  # Blank line after each board
```

File: 2025_fall_python_assignment\dawn_before_final\dijkstra_algorism\subway.py

```
import sys
import math

def get_dist(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def solve():
    # Read all input from stdin
    input_data = sys.stdin.read().split()
    if not input_data:
        return

    iterator = iter(input_data)
    try:
        home_x = int(next(iterator))
        home_y = int(next(iterator))
        school_x = int(next(iterator))
        school_y = int(next(iterator))
    except StopIteration:
        return

    nodes = []
    nodes.append((home_x, home_y)) # Index 0: Home
    nodes.append((school_x, school_y)) # Index 1: School

    # Adjacency list for subway connections
    # We will handle walking connections implicitly as they exist between all
pairs
    subway_adj = {} # Map u -> list of (v, time)

    while True:
        try:
            # Read a subway line
            line_stops = []
            while True:
                x = int(next(iterator))
                y = int(next(iterator))
                if x == -1 and y == -1:
                    break

                nodes.append((x, y))
                line_stops.append(len(nodes) - 1)

            # Add subway edges for this line
            for i in range(len(line_stops) - 1):
                u = line_stops[i]
                v = line_stops[i+1]
                d = get_dist(nodes[u], nodes[v])
                # Subway speed: 40 km/h = 40000 m / 60 min
                # Time = distance / speed = distance * 60 / 40000 = distance *
0.0015
                t = d * 0.0015

                if u not in subway_adj: subway_adj[u] = []
                if v not in subway_adj: subway_adj[v] = []

                subway_adj[u].append((v, t))
                subway_adj[v].append((u, t))

        except StopIteration:
            break

    num_nodes = len(nodes)

    # Dijkstra's Algorithm
    # Since the graph is dense (walking edges between all pairs),
    # we use the O(V^2) implementation without a priority queue.

    dist = [float('inf')] * num_nodes
    dist[0] = 0
    visited = [False] * num_nodes

    # Walking speed: 10 km/h = 10000 m / 60 min
    # Time = distance * 60 / 10000 = distance * 0.006
    WALK_FACTOR = 0.006

    for _ in range(num_nodes):
        # Find unvisited node with smallest distance
        u = -1
        min_val = float('inf')
        for i in range(num_nodes):
            if not visited[i] and dist[i] < min_val:
                min_val = dist[i]
                u = i

        if u == -1 or dist[u] == float('inf'):
            break

        visited[u] = True

        if u == 1: # Reached School
            break

        # Relax edges
        # 1. Walking edges (to all other nodes)
        for v in range(num_nodes):
            if not visited[v]:
                d = get_dist(nodes[u], nodes[v])
                new_dist = dist[u] + d * WALK_FACTOR
                if new_dist < dist[v]:
                    dist[v] = new_dist

        # 2. Subway edges (if any)
        if u in subway_adj:
            for v, t in subway_adj[u]:
                if not visited[v]:
                    if dist[u] + t < dist[v]:
                        dist[v] = dist[u] + t

    print(round(dist[1]))

if __name__ == '__main__':
    solve()
```

File: 2025_fall_python_assignment\dawn_before_final\dijkstra_algorism\walking_hills.py

```
m, n, p = map(int, input().split())
hills = list(list(map(str, input().split())) for _ in range(m))
def dijkstra(hills, m, n, start, end):
    import heapq
    if hills[start[0]][start[1]] == "#" or hills[end[0]][end[1]] == "#":
        return -1
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    priority_queue = [(0, start, int(hills[start[0]][start[1]]))]
    strength_consumption = [[float('inf')] * n for _ in range(m)]
    strength_consumption[start[0]][start[1]] = 0
    while priority_queue:
        current_strength_consumption, (x, y), last_altitude =
heapq.heappop(priority_queue)
        if (x, y) == end:
            return current_strength_consumption
        if current_strength_consumption > strength_consumption[x][y]:
            continue
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < m and 0 <= ny < n and hills[nx][ny] != "#":
                new_strength_consumption = current_strength_consumption +
abs(int(hills[nx][ny]) - last_altitude)
                if new_strength_consumption < strength_consumption[nx][ny]:
                    strength_consumption[nx][ny] = new_strength_consumption
                    heapq.heappush(priority_queue, (new_strength_consumption, (nx,
ny), int(hills[nx][ny])))

    return -1

for _ in range(p):
    start_x, start_y, end_x, end_y = map(int, input().split())
    result = dijkstra(hills, m, n, (start_x, start_y), (end_x, end_y))
    print(result if result != -1 else "NO")

'''4 5 3
0 0 0 0 0
0 1 1 2 3
# 1 0 0 0
0 # 0 0 0
0 0 3 4
1 0 1 4
3 4 3 0'''
```

File: 2025_fall_python_assignment\dawn_before_final\disjoint_set_practice\food_chains.py

```
class disjoint_set:
    def __init__(self, n):
        self.node = [i for i in range(3*n)]
        self.rank = [0 for _ in range(3*n)]

    def find(self, x):
        if self.node[x] != x:
            self.node[x] = self.find(self.node[x])
        return self.node[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return
        if self.rank[root_x] < self.rank[root_y]:
            self.node[root_x] = root_y
        else:
            self.node[root_y] = root_x
            if self.rank[root_x] == self.rank[root_y]:
                self.rank[root_x] += 1

number_of_animals, number_of_statements = map(int, input().split())
ds = disjoint_set(number_of_animals)
false_statements = 0
for _ in range(number_of_statements):
    statement_type, x, y = map(int, input().split())
    x -= 1
```

```
        y -= 1
        if x < 0 or x >= number_of_animals or y < 0 or y >= number_of_animals:
            false_statements += 1
            continue
        if statement_type == 1:
            if ds.find(x) == ds.find(y + number_of_animals) or ds.find(x) == ds.find(y
+ 2 * number_of_animals):
                false_statements += 1
            else:
                ds.union(x, y)
                ds.union(x + number_of_animals, y + number_of_animals)
                ds.union(x + 2 * number_of_animals, y + 2 * number_of_animals)
        else:
            if ds.find(x) == ds.find(y) or ds.find(x) == ds.find(y + 2 *
number_of_animals):
                false_statements += 1
            else:
                ds.union(x, y + number_of_animals)
                ds.union(x + number_of_animals, y + 2 * number_of_animals)
                ds.union(x + 2 * number_of_animals, y)

print(false_statements)
```

File: 2025_fall_python_assignment\dawn_before_final\dp_practice\Kuriyama Mirai's Stones.py

```
number_of_stones = int(input())
costs_of_stones = list(map(int, input().split()))
prefix_sums = [0]
for cost in costs_of_stones:
    prefix_sums.append(prefix_sums[-1] + cost)
costs_of_stones.sort()
non_decreasing_prefix_sums = [0]
for cost in costs_of_stones:
    non_decreasing_prefix_sums.append(non_decreasing_prefix_sums[-1] + cost)
number_of_questions = int(input())
for _ in range(number_of_questions):
    question_type, l, r = map(int, input().split())
    if question_type == 1:
        print(prefix_sums[r] - prefix_sums[l - 1])
    else:
        print(non_decreasing_prefix_sums[r] - non_decreasing_prefix_sums[l - 1])
```

File: 2025_fall_python_assignment\dawn_before_final\dp_practice\Sereja and Suffixes.py

```
n, m = map(int, input().split())
arr = list(map(int, input().split()))
distinct_numbers = set()
dp_table = [0] * (n + 1)
for i in range(n - 1, -1, -1):
    number = arr[i]
    if number in distinct_numbers:
        dp_table[i] = dp_table[i + 1]
    else:
        distinct_numbers.add(number)
        dp_table[i] = dp_table[i + 1] + 1

for _ in range(m):
    l = int(input()) - 1
    print(dp_table[l])
```

File: 2025_fall_python_assignment\dawn_before_final\last_year\adjust_temp.py

```
t = int(input())
for _ in range(t):
    l, r, x = map(int, input().split())
    a, b = map(int, input().split())
    if abs(b - r) < x and abs(b - l) < x and a != b:
        print(-1)
        continue
    if abs(l - a) < x and abs(r - a) < x and a != b:
        print(-1)
        continue
    if a == b:
        print(0)
        continue
    steps = 0
    while abs(a - b) < x:
        if abs(b - r) >= abs(b - l):
            a = r if abs(r - a) >= x else l
        else:
            a = l if abs(l - a) >= x else r
        steps += 1
    steps += 1
    print(steps)
```

File: 2025_fall_python_assignment\dawn_before_final\last_year\buy_apple.py

```
from collections import defaultdict
n, m = map(int, input().strip().split())
price_tag = list(map(int, input().strip().split()))
fruit_dict = defaultdict(int)
for _ in range(m):
    fruit = input().strip()
    fruit_dict[fruit] = fruit_dict.get(fruit, 0) + 1
min_sum = 0
max_sum = 0
nums = sorted(fruit_dict.values(), reverse=True)
price_tag.sort()
for i in range(len(nums)):
    min_sum += nums[i] * price_tag[i]

price_tag.sort(reverse=True)
for i in range(len(nums)):
    max_sum += nums[i] * price_tag[i]

print(min_sum, max_sum)
```

File: 2025_fall_python_assignment\dawn_before_final\last_year\divide_num.py

```
n, k = map(int, input().strip().split())
def dfs(n, current_step, k, largest):
    if current_step == k:
        if n == 0:
            return 1
        else:
            return 0
    if n <= 0:
        return 0
    total_ways = 0
    for i in range(largest, n+1, 1):
        total_ways += dfs(n - i, current_step + 1, k, i)
    return total_ways

print(dfs(n, 0, k, 1))
```

File: 2025_fall_python_assignment\dawn_before_final\last_year\pascal.py

```
code = list(input().strip().split(";"))
code.pop()
lis = [0] * 3
for item in code:
    var, val = item.split(":=")
    if var == "a":
        lis[0] = int(val)
    elif var == "b":
        lis[1] = int(val)
    elif var == "c":
        lis[2] = int(val)

print(" ".join(map(str, lis)))
```

File: 2025_fall_python_assignment\dawn_before_final\last_year\scholarship.py

```
t = int(input().strip())
stu_info = []
for i in range(1, t+1):
    a, b, c = map(int, input().strip().split())
    sum = a + b + c
    stu_info.append((sum, a, i))
stu_info.sort(key=lambda x: (-x[0], -x[1], x[2]))
pointer = 1
for info in stu_info:
    print(f"{info[2]} {info[0]}")
    pointer += 1
    if pointer > 5:
        break
```

File: 2025_fall_python_assignment\dawn_before_final\last_year\snake_in_maze.py

```
from collections import deque

def bfs(grid, n):
    # steps[x][y][state] -> state 0: horizontal, 1: vertical
    # We use (x, y) as the HEAD position
    steps = [[[-1, -1] for _ in range(n)] for _ in range(n)]
    queue = deque()
    # Starting state: Head at (0, 1), Horizontal (0)
    steps[0][1][0] = 0
    queue.append((0, 1, 0))

    while queue:
        x, y, state = queue.popleft()
        dist = steps[x][y][state]

        # Target: Head at (n-1, n-1) and Horizontal (0)
        if x == n-1 and y == n-1 and state == 0:
            return dist

        if state == 0: # HORIZONTAL
            # 1. Move Right
            nx, ny = x, y + 1
            if ny < n and grid[nx][ny] == 0 and steps[nx][ny][0] == -1:
                steps[nx][ny][0] = dist + 1
                queue.append((nx, ny, 0))
            # 2. Move Down (Must check both cells occupied by snake)
            nx, ny = x + 1, y
            if nx < n and grid[nx][ny] == 0 and grid[nx][ny-1] == 0 and steps[nx]
[ny][0] == -1:
                steps[nx][ny][0] = dist + 1
                queue.append((nx, ny, 0))
            # 3. Rotate Clockwise (Check the 2x2 area)
            if x + 1 < n and grid[x+1][y] == 0 and grid[x+1][y-1] == 0 and
steps[x+1][y-1][1] == -1:
                steps[x+1][y-1][1] = dist + 1
                queue.append((x+1, y-1, 1))

        else: # VERTICAL
            # 1. Move Down
            nx, ny = x + 1, y
            if nx < n and grid[nx][ny] == 0 and steps[nx][ny][1] == -1:
                steps[nx][ny][1] = dist + 1
                queue.append((nx, ny, 1))
            # 2. Move Right (Must check both cells occupied by snake)
            nx, ny = x, y + 1
            if ny < n and grid[nx][ny] == 0 and grid[nx-1][ny] == 0 and steps[nx]
[ny][1] == -1:
                steps[nx][ny][1] = dist + 1
                queue.append((nx, ny, 1))
            # 3. Rotate Counter-Clockwise (Check the 2x2 area)
            if y + 1 < n and grid[x][y+1] == 0 and grid[x-1][y+1] == 0 and
steps[x-1][y+1][0] == -1:
                steps[x-1][y+1][0] = dist + 1
                queue.append((x-1, y+1, 0))

    return -1

# Input handling
import sys
input_data = sys.stdin.read().split()
if not input_data:
    exit()
n = int(input_data[0])
grid = []
idx = 1
for r in range(n):
    grid.append([int(x) for x in input_data[idx:idx+n]])
    idx += n

print(bfs(grid, n))
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam2\biggest_integer.py

```
import sys
from functools import cmp_to_key

# Custom comparator: returns -1 if x should come before y
def compare(x, y):
    if x + y > y + x:
        return -1
    elif x + y < y + x:
        return 1
    else:
        return 0

# Read inputs
# Using sys.stdin.read().split() is often safer for competitive programming
# to handle all whitespace/newlines automatically
input_data = sys.stdin.read().split()
if not input_data:
    exit()

iterator = iter(input_data)
m = int(next(iterator))
n = int(next(iterator))

number_list = []
for _ in range(n):
    number_list.append(next(iterator))

# Sort using the custom comparator
# This ensures that for any subset we pick, the relative order is optimal
number_list.sort(key=cmp_to_key(compare))

# DP Initialization
# dp[i] = max value with exactly i digits
dp = [-1] * (m + 1)
dp[0] = 0
```

```
for num_str in number_list:
    length = len(num_str)
    num_val = int(num_str)
    # Iterate backwards to avoid using the same number multiple times for one
state
    for j in range(m, length - 1, -1):
        if dp[j - length] != -1:
            # Append current number to the end of the best previous number
            new_val = dp[j - length] * (10 ** length) + num_val
            if new_val > dp[j]:
                dp[j] = new_val

# Find the largest value among all valid lengths
# Since they are integers, a longer number is always larger.
# So we just look from m down to 0.
for j in range(m, -1, -1):
    if dp[j] != -1:
        print(dp[j])
        break
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam2\easy_calculation.py

```
def cal_how_many_power(n):
    power = 0
    current = 1
    while current <= n:
        current *= 2
        power += 1
    return power - 1

def cal_strange_sequence(n):
    return (n + 1) * n // 2 - 2 * (2 ** (cal_how_many_power(n)+1) - 1)

t = int(input())
for _ in range(t):
    n = int(input())
    print(cal_strange_sequence(n))
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam2\get_code.py

```
from collections import deque
def cal_power(n):
    power = 0
    current = 1
    while current <= n:
        power += 1
        current *= 2
    return power - 1

string = list(input())
n = len(string)
buffer_code = deque()
for i in range(cal_power(n) + 1):
    buffer_code.append(string[2**i - 1])
output = ''
while len(buffer_code) > 1:
    first = buffer_code.popleft()
    second = buffer_code.pop()
    output += first + second

if buffer_code:
    output += buffer_code.popleft()

print(output)
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam2\jump_in_matrix.py

```
def find_the_shortest_path(start, target):
    from collections import deque
    queue = deque()
    queue.append((start, 0, ''))
    visited = set()
    while queue:
        position, steps, path = queue.popleft()
        if position == target:
            return steps, path
        if position in visited:
            continue
        visited.add(position)
        # in the house
        queue.append((position * 3, steps + 1, path + 'H'))
        # out of the house
        queue.append((position // 2, steps + 1, path + 'O'))
    return -1, ''
while True:
    start, target = map(int, input().split())
    if start == 0 and target == 0:
        break
    else:
        steps, path = find_the_shortest_path(start, target)
        print(steps)
        print(path)
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam2\water_plants.py

```
n, a, b = map(int, input().split())
plants = list(map(int, input().split()))
alice_pointer = 0
bob_pointer = n - 1
alice_water = a
bob_water = b
alice_count = 0
bob_count = 0
while alice_pointer < bob_pointer:
    if alice_water >= plants[alice_pointer]:
        alice_water -= plants[alice_pointer]
        alice_pointer += 1
    else:
        alice_count += 1
        alice_water = a - plants[alice_pointer]
        alice_pointer += 1
    if bob_water >= plants[bob_pointer]:
        bob_water -= plants[bob_pointer]
        bob_pointer -= 1
    else:
        bob_count += 1
        bob_water = b - plants[bob_pointer]
        bob_pointer -= 1
if alice_pointer == bob_pointer:
    if max(alice_water, bob_water) < plants[alice_pointer]:
        alice_count += 1

print(alice_count + bob_count)
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam2\when_cow_learn_alphabet.py

```
n = int(input())
vocab = [input().strip() for _ in range(4)]
for _ in range(n):
    word = input().strip()
    visited = [False] * 4
    def backtrack(index):
        if index == len(word):
            return True
        for i in range(4):
            if not visited[i] and word[index] in vocab[i]:
                visited[i] = True
                if backtrack(index + 1):
                    return True
                visited[i] = False
        return False
    if backtrack(0):
        print("YES")
    else:
        print("NO")
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam3\CPU_usage.py

```
n = int(input())
progress_info = []
for _ in range(n):
    compute_time, write_time = map(int, input().split())
    progress_info.append((write_time, compute_time))
progress_info.sort(reverse = True)
total_compute_time = 0
total_write_time = 0
for write_time, compute_time in progress_info:
    total_compute_time += compute_time
    total_write_time = max(total_write_time, total_compute_time + write_time)

print(total_write_time)
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam3\crab_grab_mushrooms.py

```
from collections import deque

n = int(input())
map = [list(map(int, input().split())) for _ in range(n)]
crab_position = []
for i in range(n):
    for j in range(n):
        if map[i][j] == 9:
            target = (i, j)
        if map[i][j] == 5:
            crab_position.append((i, j))

def bfs(start1, start2, target):
    queue = deque()
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    visited = set()
    queue.append((start1, start2))
    visited.add(tuple(sorted((start1, start2))))
    while queue:
        curr1, curr2 = queue.popleft()
        if curr1 == target or curr2 == target:
            return True
        for dx, dy in directions:
            new1 = (curr1[0] + dx, curr1[1] + dy)
            new2 = (curr2[0] + dx, curr2[1] + dy)
            if 0 <= new1[0] < n and 0 <= new1[1] < n and map[new1[0]][new1[1]] !=
1:
                if 0 <= new2[0] < n and 0 <= new2[1] < n and map[new2[0]][new2[1]]
!= 1:
                    state = tuple(sorted((new1, new2)))
                    if state not in visited:
                        visited.add(state)
                        queue.append((new1, new2))
    return False

if bfs(crab_position[0], crab_position[1], target):
    print("yes")
else:
    print("no")

'''0 0 0 0 0 9
0 0 1 0 1 1
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 1 0 0
0 0 0 1 5 5'''
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam3\falling_board.py

```
h, l, n = map(int, input().split())
velocity_list = list(map(int, input().split()))
velocity_list.sort()
if n % 2 == 0:
    must_get = n // 2 + 1
```

```
else:
    must_get = n - n // 2
split_velocity = velocity_list[must_get - 1]
reach_time = l / split_velocity
height = h - reach_time ** 2 * 5
print(f"{height:.2f}")
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam3\onion_and_matrix.py

```
def calcu_one_layer_matrix(matrix):
    if len(matrix) == 2 and len(matrix[0]) == 2:
        return [sum(matrix[0]) + sum(matrix[1])]
    if len(matrix) == 1 and len(matrix[0]) == 1:
        return [matrix[0][0]]
    else:
        peeled_matrix = [matrix[i][1:len(matrix)-1] for i in range(1,
len(matrix)-1)]
        calcu_sum = 0
        calcu_sum += sum(matrix[0]) + sum(matrix[-1])
        for i in range(1, len(matrix)-1):
            calcu_sum += matrix[i][0] + matrix[i][-1]
        return [calcu_sum] + calcu_one_layer_matrix(peeled_matrix)

n = int(input())
matrix = []
for _ in range(n):
    row = list(map(int, input().split()))
    matrix.append(row)

result = calcu_one_layer_matrix(matrix)
print(max(result))
'''5
1 0 1 0 1
0 1 1 1 0
0 1 7 1 0
0 1 1 1 0
1 0 1 0 1'''
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam3\reverse_binary_representation.py

```
num = int(input())
string = bin(num)[2:]
```

```
reversed_string = string[::-1]
if reversed_string == string:
    print("Yes")
else:
    print("No")
```

File: 2025_fall_python_assignment\dawn_before_final\mock_exam3\rose_red_and_blue.py

```
original_string_list = list(input().strip())
string_list = original_string_list.copy()
count = 0
group_number = 1
state = string_list[0]
for i in range(1, len(string_list)):
    if string_list[i] != state:
        if i == len(string_list) - 1:
            if string_list[i] == 'B':
                count += 1
                continue
            state = string_list[i]
            group_number += 1
            continue
        if string_list[i+1] == state:
            count += 1
            continue
        else:
            group_number += 1
            state = string_list[i]
    else:
        continue

if state == 'R':
    count += (group_number - 1)
else:
    count += group_number

print(count)
```

File: 2025_fall_python_assignment\dawn_before_final\problems_since_Tuesday\double_eleven.py

```
import sys
```

```
# Increase recursion depth just in case, though n is small
sys.setrecursionlimit(2000)

def solve():
    # Read n and m
    line = sys.stdin.readline()
    while line and not line.strip(): # Skip empty leading lines if any
        line = sys.stdin.readline()

    if not line:
        return

    try:
        n, m = map(int, line.split())
    except ValueError:
        return

    # Read items
    items = []
    for _ in range(n):
        line = sys.stdin.readline().strip()
        parts = line.split()
        item_options = []
        for part in parts:
            if ':' in part:
                shop_str, price_str = part.split(':')
                item_options.append((int(shop_str), int(price_str)))
        items.append(item_options)

    # Read coupons
    # coupons[shop_id] = list of (q, x)
    coupons = {}
    for i in range(1, m + 1):
        line = sys.stdin.readline().strip()
        shop_coupons = []
        if line:
            parts = line.split()
            for part in parts:
                if '-' in part:
                    q_str, x_str = part.split('-')
                    shop_coupons.append((int(q_str), int(x_str)))
        coupons[i] = shop_coupons

    min_total_cost = float('inf')

    # shop_totals[i] stores the total price of items selected from shop i
    shop_totals = [0] * (m + 1)

    def dfs(item_idx):
        nonlocal min_total_cost

        if item_idx == n:
            # Calculate final cost
            total_marked_price = sum(shop_totals)
            # 1. Cross-shop discount
            # "每满300，可以减去50"
            cross_shop_discount = (total_marked_price // 300) * 50

            # 2. Shop coupons
            shop_discounts = 0
            for s in range(1, m + 1):
                current_shop_total = shop_totals[s]
                best_coupon_cut = 0
                for q, x in coupons[s]:
                    if current_shop_total >= q:
                        if x > best_coupon_cut:
                            best_coupon_cut = x
                shop_discounts += best_coupon_cut

            final_cost = total_marked_price - cross_shop_discount - shop_discounts

            if final_cost < min_total_cost:
                min_total_cost = final_cost
            return

        # Try buying current item from each available shop
        for shop_id, price in items[item_idx]:
            shop_totals[shop_id] += price
            dfs(item_idx + 1)
            shop_totals[shop_id] -= price # Backtrack

    dfs(0)
    print(min_total_cost)

if __name__ == '__main__':
    solve()
```

File: 2025_fall_python_assignment\dawn_before_final\problems_since_Tuesday\double_eleven2.py

```
n, m = map(int, input().split())
commodity_infos = [[] for _ in range(n)]
coupon_infos = [[] for _ in range(m)]
for j in range(n):
    information = input().split()
    for i in information:
        shop_id, price = map(int, i.split(':'))
        commodity_infos[j].append((shop_id - 1, price))
for j in range(m):
    coupon_info = input().split()
    for coupon in coupon_info:
        q, x = map(int, coupon.split('-'))
        coupon_infos[j].append((q, x))

total_min_cost = float('inf')
cost_list_for_each_shop = [0] * m
def dfs(item_index):
    global total_min_cost
    if item_index == n:
        total_price = sum(cost_list_for_each_shop)
        cross_shop_discount = (total_price // 300) * 50
        shop_discount = 0
        for shop_index in range(m):
            current_shop_total = cost_list_for_each_shop[shop_index]
            best_coupon_cut = 0
            for q, x in coupon_infos[shop_index]:
                if current_shop_total >= q:
                    if x > best_coupon_cut:
                        best_coupon_cut = x
            shop_discount += best_coupon_cut
        final_price = total_price - cross_shop_discount - shop_discount
        if final_price < total_min_cost:
            total_min_cost = final_price
        return
    for shop_index in range(m):
        for shop_id, price in commodity_infos[item_index]:
            if shop_id == shop_index:
                cost_list_for_each_shop[shop_index] += price
                dfs(item_index + 1)
                cost_list_for_each_shop[shop_index] -= price

dfs(0)
print(total_min_cost)

'''2 2
1:100 2:120
1:300 2:350
200-30 400-70
100-80'''
```

File: 2025_fall_python_assignment\dawn_before_final\recap_makes_perfect\pile_pig.py

```
import sys
lines = sys.stdin.read().strip().split('\n')
stack = []
min_stack = []
for line in lines:
    if len(line) != 3:
        a, b = map(str, line.strip().split())
```

```
            b = int(b)
            stack.append(b)
            if not min_stack or b <= min_stack[-1]:
                min_stack.append(b)

        elif line == 'pop':
            if stack:
                top = stack.pop()
                if top == min_stack[-1]:
                    min_stack.pop()
        elif line == 'min':
            if stack:
                print(min_stack[-1])
```

File: 2025_fall_python_assignment\dawn_before_final\some_important_implementation\sieve_e.py

```python
def sieve_of_eratosthenes(limit):
    # Create a boolean array "prime[0..limit]" and initialize
    # all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    primes = [True] * (limit + 1)
    primes[0] = primes[1] = False  # 0 and 1 are not prime numbers

    p = 2
    while (p * p <= limit):
        # If primes[p] is not changed, then it is a prime
        if primes[p] == True:
            # Update all multiples of p starting from p*p
            # We start at p*p because smaller multiples
            # have already been marked by smaller primes
            for i in range(p * p, limit + 1, p):
                primes[i] = False
        p += 1

    # Extract the prime numbers from the boolean list
    return [num for num, is_prime in enumerate(primes) if is_prime]

# Example usage:
print(sieve_of_eratosthenes(30))
# Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

File: 2025_fall_python_assignment\dawn_before_final\stack_structure\poland_note.py

```
operator = list(input().split(' '))
operator.reverse()
stack = []
for op in operator:
    if op == '+':
        a = stack.pop()
        b = stack.pop()
        stack.append(a + b)
    elif op == '-':
        a = stack.pop()
        b = stack.pop()
        stack.append(a - b)
    elif op == '*':
        a = stack.pop()
        b = stack.pop()
        stack.append(a * b)
    elif op == '/':
        a = stack.pop()
        b = stack.pop()
        stack.append(a / b)
    else:
        stack.append(float(op))
print(f'{stack[0]:.6f}')
```

File: 2025_fall_python_assignment\dawn_before_final\stack_structure\probable_out_sequence.py

```
from itertools import permutations
n = int(input())
lis = [i for i in range(1, n + 1)]
all_sequence = list(permutations(lis))
valid_sequence = []
for sequence in all_sequence:
    stack = []
    current = 0
    for num in range(1, n + 1):
        stack.append(num)
        while stack and stack[-1] == sequence[current]:
            stack.pop()
            current += 1
    if current == n:
        valid_sequence.append(sequence)
for seq in valid_sequence:
    print(' '.join(map(str, seq)))
```

File: 2025_fall_python_assignment\dawn_before_final\stack_structure\reversed_poland_note.py

```python
expression = list(input().split())
# give you a normal expression, you need to convert it to reversed polish note
def precedence(op):
    if op == '+' or op == '-':
        return 1
    if op == '*' or op == '/':
        return 2
    return 0
def infix_to_postfix(expression):
    stack = []
    output = []
    for token in expression:
        if token.isalnum():  # If the token is an operand (number/variable)
            output.append(token)
        elif token == '(':  # If the token is '(', push it to the stack
            stack.append(token)
        elif token == ')':  # If the token is ')', pop and output from the stack
until an '(' is encountered
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # Pop the '(' from the stack
        else:  # The token is an operator
            while (stack and precedence(stack[-1]) >= precedence(token)):
                output.append(stack.pop())
            stack.append(token)
    # Pop all the operators from the stack
    while stack:
        output.append(stack.pop())
    return ' '.join(output)

print(infix_to_postfix(expression))
```

File: 2025_fall_python_assignment\dawn_before_final\stack_structure\valid_out_stack_sequence.py

```python
n = int(input())
sequence = list(map(int, input().split()))
# judge whether the sequence can be a valid stack pop sequence
stack = []
index = 0
for i in range(1, n + 1):
    stack.append(i)
    while stack and stack[-1] == sequence[index]:
        stack.pop()
        index += 1

if index == n:
    print("Yes")
else:
    print("No")
```

动态规划计数类问题思维模板

本模板围绕 **“约束 → 状态 → 转移 → 初始化”** 四步展开，旨在解决计数类问题中常见的“重复计数”和“子问题拆分”难点。

一、 通用思维框架（四步走）

1. 第一步：给问题加 “有序约束” —— 解决重复计数

计数类问题常因“组合无序”导致重复（如 $1+3$ 和 $3+1$）。核心是给组合加一个**“不可逆约束”**，让每个组合只对应一种唯一的表达序列。

- **常见约束方向：**
  - **约束“元素上限”**：如“最大数不超过 $j$”（适配整数划分、组合总和）。
  - **约束“元素个数”**：如“恰好分成 $k$ 个元素”（适配将 $n$ 分成 $k$ 个整数）。
  - **约束“元素顺序”**：如“按非递增排列”（避免乱序重复）。
- **选择原则**：选“能让子问题最容易拆分”的。

2. 第二步：定义状态 —— 建立子问题联系

状态格式固定为：`dp[i][j]` = **符合约束的、由 $i$ 和 $j$ 限定的子问题的解。**

- **$i$（目标总量）**：通常是目标和、总金额或被划分的总数。
- **$j$（约束变量）**：通常是最大允许数、可选元素索引或限制个数。

3. 第三步：推导状态转移方程 —— “分情况讨论”

通过将当前问题拆分成两个 **互斥且完备** 的子问题，确保不重不漏：

- **常用拆分逻辑：**
  - **情况 A**：不使用当前约束/元素 $j$。
  - **情况 B**：至少使用一次当前约束/元素 $j$。
- **通用公式**：$dp[i][j] = dp[\text{情况 A}] + dp[\text{情况 B}]$

4. 第四步：确定初始化和边界 —— 设定起点

- **总量为 0**：`dp[0][j] = 1`（凑成和为 0 只有一种方案：什么都不选）。
- **约束为 0**：`dp[i][0] = 0`（当 $i > 0$ 时，没有元素可选则方案数为 0）。
- **无效约束**：当 $i < j$ 时，约束超过总量，通常有 $dp[i][j] = dp[i][i]$。

二、 5 个经典题型实战

题型	状态定义 `dp[i][j]`	状态转移方程
1. 整数划分	将 i 划分成最大数 $\le j$ 的方案数	$dp[i][j] = dp[i][j-1] + dp[i-j][j]$
2. 组合总和	用前 j 个元素凑出和为 i (可重复)	$dp[i][j] = dp[i][j-1] + dp[i-nums[j-1]][j]$
3. 凑硬币计数	用面额 $\le coins[j]$ 凑出金额 i	$dp[i][j] = dp[i][j-1] + dp[i-coins[j]][j]$
4. 无重复组合	用前 j 个元素凑出和为 i (不可重复)	$dp[i][j] = dp[i][j-1] + dp[i-nums[j-1]][j-1]$
5. n 分成 k 个数	将 i 恰好分成 j 个正整数	$dp[i][j] = dp[i-1][j-1] + dp[i-j][j]$

三、 关键技巧与避坑

1. **空间优化（降维）**：
   - 如果状态转移只依赖于 `dp[...][j-1]`，可以将二维数组压缩为一维。
   - **完全背包类（可重复）**：从小到大遍历 i。
   - **0-1 背包类（不可重复）**：从大到小遍历 i。
2. **边界校验**：
   - 在计算 $dp[i-j]$ 或 $dp[i-nums[j]]$ 前，务必判断 $i \ge \text{cost}$，防止数组下标越界。
3. **验证逻辑**：
   - 手动模拟一个小样本（如 $n=4$ 的整数划分），确保初始化 `dp[0]` 的传导逻辑能正确产生预期的组合数。