▶ScoreOfCells

```
#include <iostream>
#include <vector>
using namespace std;
void Solve(int i, int j, vector<vector<int>> &table, vector<vector<int>> &count)
{
  int n = table.size(), m = table[0].size();
  if (i >= n || j >= m)
    return;
  // For down
  if (i + 1 < n && table[i + 1][j] >= table[i][j])
  {
    count[i + 1][j]++;
    Solve(i + 1, j, table, count);
  }
  // For right
  if (j + 1 < m && table[i][j + 1] >= table[i][j])
  {
    count[i][j + 1]++;
    Solve(i, j + 1, table, count);
  }
}
int main()
{
  int n, m, k;
  cin >> n >> m;

  // Read the table
  vector<vector<int>> table(n, vector<int>(m));
  for (int i = 0; i < n; i++)
```

```
{
    for (int j = 0; j < m; j++)
    {
        cin >> table[i][j];
    }
}

// Read the value of k
cin >> k;
vector<vector<int>> count(n, vector<int>(m, 0));
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        Solve(i, j, table, count);
    }
}
bool found = false;
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        if (count[i][j] == k)
        {
            cout << i << " " << j << endl;
            found = true;
        }
    }
}
if (!found)
    cout << "NO";
    return 0;
}
```

▶**PlaceFinder**

```python
import math
from collections import defaultdict, deque


def connect(adjclist, device1, device2, distance, angle):
    # Calculate the x and y components of the distance
    x = distance * math.cos(math.radians(angle))
    y = distance * math.sin(math.radians(angle))
    adjclist[device1].append((device2, x, y))
    adjclist[device2].append((device1, -x, -y))  # Reverse the direction for undirected connection


def finddist(adjclist, start, target):
    # Perform BFS to find the shortest path and calculate the distance
    q = deque([(start, 0.0, 0.0)])  # (current node, x sum, y sum)
    vis = set([start])

    while q:
        curr, sumxval, sumyval = q.popleft()
        if curr == target:
            return math.sqrt(sumxval**2 + sumyval**2)  # Correct the formula for distance

        for adj, dx, dy in adjclist[curr]:
            if adj not in vis:
                vis.add(adj)
                q.append((adj, sumxval + dx, sumyval + dy))
    return -1  # Return -1 if no path exists
```

```
# Main input section
n = int(input().strip())  # Number of devices
devices = input().strip().split()  # List of device names

# Create the adjacency list for the connections
adjclist = defaultdict(list)

for _ in range(n):
    devid = int(input().strip())  # Device ID (1-indexed)
    conn = int(devices[devid - 1].split(':')[1])  # Extract number of connections
    for _ in range(conn):
        nid, dist, angle = map(int, input().strip().split())  # Neighbour ID, distance, and angle
        connect(adjclist, devid, nid, dist, angle)

# Start and end device IDs
start, end = map(int, input().strip().split())

# Find the distance using BFS
distance = finddist(adjclist, start, end)

# Output the result
if distance != -1:
    print(f"{distance:.2f}")
else:
    print("Path not found")
```

▶HammingDistance

```
def min_cost_hamming_distance(s, A, B):
    if not all(c in '01' for c in s):
        return "INVALID"

    cost = 0
    for i in range(len(s) - 1):
        if s[i] != s[i+1]:
            cost += A if s[i] == '0' else B

    # Rearrange the string
    new_s = ''.join(sorted(s))

    # Calculate Hamming distance
    hamming_distance = sum(1 for i in range(len(s)) if s[i] != new_s[i])

    return hamming_distance

# Main function to handle multiple test cases
def main():
    T = int(input())  # Number of test cases
    for _ in range(T):
        s = input()  # Binary string
        A, B = map(int, input().split())  # Costs A and B
        result = min_cost_hamming_distance(s, A, B)
        print(result)

# Corrected check for script execution
if __name__ == "__main__":
    main()
```

▶**HarmonicHomology**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#define MAX 256
typedef struct Node
{
    char name[50];
    int level;
    struct Node* children[MAX];
    int num_children;
} Node;
Node* create_node(const char* name)
{
    Node* node = (Node*)malloc(sizeof(Node));
    strcpy(node->name, name);
    node->num_children = 0;
    return node;
}
void add_child(Node* parent, Node* child)
{
    parent->children[parent->num_children++] = child;
}
Node* find_node(Node* nodes[], int* node_count, char* name)
{
    for (int i = 0; i < *node_count; i++)
    {
        if (strcmp(nodes[i]->name, name) == 0)
            return nodes[i];
    }
    nodes[*node_count] = create_node(name);
    return nodes[(*node_count)++];
}
void bfs_assign_levels(Node* root)
{
    Node* queue[MAX];
    int front = 0, rear = 0;
    root->level = 0;
    queue[rear++] = root;
    while (front < rear) {
        Node* current = queue[front++];
        for (int i = 0; i < current->num_children; i++)
        {
            Node* child = current->children[i];
            child->level = current->level + 1;
            queue[rear++] = child;
        }
    }
}
int main()
{
    int N, A, B, C, m, n;
    char line[1024], *token;
    Node* nodes[MAX];
    int node_count = 0;
    Node* root = NULL;
    scanf("%d\n", &N);
    for (int i = 0; i < N; i++)
    {
        fgets(line, sizeof(line), stdin);
        char* colon = strchr(line, ':');
        if (colon)
        {
            *colon = '\0';
            Node* parent = find_node(nodes, &node_count, strtok(line, " \n"));
            if (i == 0) root = parent;
            token = strtok(colon + 1, " \n");
            while (token)
            {
                Node* child = find_node(nodes, &node_count, token);
                add_child(parent, child);
                token = strtok(NULL, " \n");
            }
        }
        else
        {
            if (i == 0) root = find_node(nodes, &node_count, strtok(line, " \n"));
        }
    }
    bfs_assign_levels(root);
    char melody1[MAX][50], melody2[MAX][50];
    fgets(line, sizeof(line), stdin);
    m = 0;
    token = strtok(line, "-\n");
    while (token)
    {
        strcpy(melody1[m++], token);
        token = strtok(NULL, "-\n");
    }
    fgets(line, sizeof(line), stdin);
    n = 0;
    token = strtok(line, "-\n");
    while (token)
```

```
{
    strcpy(melody2[n++], token);
    token = strtok(NULL, "-\n");
}
scanf("%d %d %d", &A, &B, &C);
int dp[m + 1][n + 1];
memset(dp, 0, sizeof(dp));
for (int i = 1; i <= m; i++)
    dp[i][0] = dp[i - 1][0] - C;
for (int j = 1; j <= n; j++)
    dp[0][j] = dp[0][j - 1] - C;
for (int i = 1; i <= m; i++)
{
    for (int j = 1; j <= n; j++)
    {
        dp[i][j] = dp[i - 1][j] - C;
        dp[i][j] = (dp[i][j - 1] - C > dp[i][j]) ? dp[i][j - 1] - C : dp[i][j];
        int tune1_level = find_node(nodes, &node_count, melody1[i - 1])->level;
        int tune2_level = find_node(nodes, &node_count, melody2[j - 1])->level;
        if (strcmp(melody1[i - 1], melody2[j - 1]) == 0 || tune1_level == tune2_level)
        {
            dp[i][j] = (dp[i - 1][j - 1] + A > dp[i][j]) ? dp[i - 1][j - 1] + A : dp[i][j];
        }
        else
        {
            dp[i][j] = (dp[i - 1][j - 1] - B > dp[i][j]) ? dp[i - 1][j - 1] - B : dp[i][j];
        }
    }
}
printf("%d\n", dp[m][n]);
return 0;
```

```
}
```