2022 연세대학교 미래캠퍼스 슬기로운 코딩생활

A번 - 영수증

영수증에는 구매한 각 물건의 가격과 개수, 구매한 물건의 총 금액이 있습니다.

이 문제는 구매한 물건의 가격과 개수로 계산한 총 금액이 영수증에 적힌 구매한 물건의 총 금액과 일치하는지 확인하는 문제입니다.

각 물건의 가격과 개수를 곱한 값을 모두 더한 값이 영수증에 적힌 구매한 물건의 총 금액과 일치하는지 확인하면 됩니다.

C++

Java

Python

소스 1

```cpp
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    int n;
    cin >> n;
    int sum = 0;
    for (int i=0; i<n; i++) {
        int a, b;
        cin >> a >> b;
        sum += a*b;
    }
    cout << (sum == x ? "Yes" : "No") << endl;
    return 0;
}

```

B번 - 커트라인

N명이 참가한 대회에서 점수가 높은 상위 k명은 상을 받게 됩니다. 여기서 상을 받는 커트라인을 구하는 문제입니다. 문제에서 커트라인은 상을 받는 사람 중 점수가 가장 낮은 사람의 점수를 의미한다고 합니다. 즉, 커트라인은 점수가 k번째로 높은 사람의 점수를 의미하게 됩니다.

따라서, 이 문제는 입력으로 주어진 점수를 정렬한 다음, k번째로 높은 점수를 찾으면 됩니다.

C++

Java

소스 2

Python

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (int i=0; i<n; i++) {
        cin >> a[i];
    }
    sort(a.begin(), a.end());
    cout << a[n-k] << endl;
    return 0;
}

```

입력으로 주어진 점수를 `a`에 저장했고, 이를 오름차순 정렬했습니다. 1번째로 점수가 높은 사람의 점수는 `a[n-1]`입니다. 2번째로 높은 사람의 점수는 `a[n-2]`입니다. 그럼 `k`번째로 점수가 높은 사람의 점수는 `a[n-k]`임을 알 수 있습니다.

만약, `a`를 내림차순 정렬했다면, `k`번째 점수가 높은 사람의 점수는 `a[k-1]`에 있게됩니다. 소스 3은 내림차순 정렬을 한 경우입니다.

C++

Java

Python

소스 3

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (int i=0; i<n; i++) {
        cin >> a[i];
    }
    sort(a.rbegin(), a.rend());
    cout << a[k-1] << endl;
    return 0;
}

```

Java에서 `Collections.reverseOrder()`을 사용하기 위해 `a`의 선언을 소스 2와 다르게 `Integer`로 변경했습니다.

소스 2와 3 모두 시간 복잡도는 $O(N \lg N)$ 정렬을 사용하고 있기 때문에, $O(N \lg N)$ 입니다.

N의 최댓값은 $1\,000$이기 때문에, $O(N^2)$ 정렬을 사용해도 됩니다.

C번 - 연속 XOR

A부터 B까지 모든 자연수를 XOR 연산한 값을 구하는 문제입니다. XOR 연산은 $\oplus$로 표시하겠습니다.

소스 4와 같이 해결할 수 있을 것 같지만, 다음과 같이 구현하면 시간 초과를 받게 됩니다.

소스 4

C++
Java
Python

```cpp
#include <iostream>
using namespace std;
int main() {
    long long a, b;
    cin >> a >> b;
    long long ans = 0;
    for (long long i=a; i<=b; i++) {
        ans ^= i;
    }
    cout << ans << endl;
}

```

이유는 A와 B의 제한 때문입니다.

- $1 \le A \le B \le 1\,000\,000\,000\,000\,000\,000 = 10^{18}$

소스 4의 시간 복잡도는 $O(B-A+1)$로 계산할 수 있는데, 만약 $A=1$, $B=10^{18}$이라면, 10^{18}개의 $\oplus$ 연산을 수행해야 하기 때문에, 시간이 매우 오래 걸리게 됩니다.

이 문제를 해결하려면 $\oplus$의 성질 하나를 알아야 합니다.

1. $A \oplus A = 0$

$0 \oplus 0 = 0$이고, $1 \oplus 1 = 0$ 입니다. 같은 수를 $\oplus$연산하면 모든 비트가 같기 때문에, 그 결과는 0이 됩니다.

만약 1부터 $A-1$까지 모든 자연수를 $\oplus$ 연산한 결과를 SA라고 하고 1부터 B까지 모든 자연수를 $\oplus$ 연산한 결과를 SB라고 해봅시다. SA와 SB를 이용해서 A부터 B까지 모든 자연수를 $\oplus$한 결과를 구할 수 있습니다. 정답은 바로 $SA \oplus SB$ 입니다.

SA와 SB를 $\oplus$연산하게 되면, 1부터 $A-1$까지 자연수는 두 번씩 $\oplus$한 것과 같습니다. 따라서, A부터 B까지 자연수를 $\oplus$한 결과와 같게 됩니다.

이 아이디어를 이용해 소스 5를 구현할 수 있습니다.

소스 5

C++
Java
Python

```cpp
#include <iostream>
using namespace std;
long long f(long long n) {
    long long ans = 0;
    for (long long i=1; i<=n; i++) {
        ans ^= i;
    }
    return ans;
}
```

```
int main() {
    long long a, b;
    cin >> a >> b;
    cout << (f(a-1) ^ f(b)) << endl;
}

```

소스 5의 `f(n)`은 1부터 `n`까지 ⊕한 결과를 구하는 함수입니다. 소스 4와 마찬가지로 수를 하나씩 ⊕하는 것이니 시간 복잡도도 소스 4와 같습니다.

`f(1)`부터 `f(15)`까지 순서대로 값을 출력해보면 특이한 규칙을 찾을 수 있습니다.

n	f(n)	n의 이진수 표현	f(n)의 이진수 표현
1	1	0001	0001
2	3	0010	0011
3	0	0011	0000
4	4	0100	0100
5	1	0101	0001
6	7	0110	0111
7	0	0111	0000
8	8	1000	1000
9	1	1001	0001
10	11	1010	1011
11	0	1011	0000
12	12	1100	1100
13	1	1101	0001
14	15	1110	1111
15	0	1111	0000

`n`을 4로 나눈 나머지에 따라서 다음과 같은 규칙을 갖는 것을 확인할 수 있습니다.

- 4로 나눈 나머지가 0: `n`
- 4로 나눈 나머지가 1: `1`
- 4로 나눈 나머지가 2: `n+1`
- 4로 나눈 나머지가 3: `0`

이를 이용해 소스 6을 구현할 수 있습니다.

C++

Java

Python

소스 6

```cpp
#include <iostream>
using namespace std;
long long f(long long n) {
    int x = (int)(n % 4);
    if (x == 0) {
        return n;
    } else if (x == 1) {
        return 1;
    } else if (x == 2) {
        return n+1;
    } else {
        return 0;
    }
}
int main() {
    long long a, b;
    cin >> a >> b;
    cout << (f(a-1) ^ f(b)) << endl;
    return 0;
}

```

D번 - 시루의 백화점 구경

문제의 조건을 정리해보겠습니다.

1. 백화점은 세로 길이가 N, 가로 길이가 M인 격자 형태
2. 상하좌우로 인접한으로 이동할 때마다 1만큼의 체력을 소모
3. 기둥이 있는 칸으로는 이동할 수 없음
4. 마네킹과의 거리가 K 이하인 칸은 이동할 수 없음
5. 시루가 있는 위치에서 출발해 의자 중 하나에 도착할 때 소모하는 체력의 최솟값을 구해야 함

격자에서 이동하는 문제이고, 이때 이동할때 비용이 1입니다. 이러한 경우 문제에서 구해야 하는 값이 비용의 최솟값이라면 BFS 알고리즘으로 해결할 수 있습니다.

4번 조건이 없다면 시루가 있는 위치를 시작점으로, 의자의 위치를 도착점으로 하는 BFS로 해결할 수 있습니다. 이때 시간 복잡도는 $O(NM)$이 됩니다.

이제 4번 조건이 있는 경우를 생각해봅시다. 어떤 칸으로 이동할 수 있는지 없는지 알아볼때마다 그 칸과 모든 마네킹과의 거리를 구하고, K 이하가 아닌지 확인해야 합니다. 어떤 칸과 모든 마네킹과의 거리가 K이하가 아닌지 확인하려면, 그 칸과 가장 가까운 마네킹과의 거리만 구해도 됩니다.

마네킹은 최대 $NM-2$개 있을 수 있으니, 어떤 칸을 이동하려고 할 때마다 가장 가까운 마네킹과의 거리를 구하게 BFS를 구현하면 $O(NMNM)$이 됩니다. $N, M \le 2\,000$이니 시간 초과를 피할 수 없습니다.

격자판에서 A에서 B까지의 거리는 B에서 A까지의 거리와 같습니다. 마네킹은 이동하지 않으니 어떤 칸과 가장 가까운 마네킹과의 거리도 변하지 않습니다. 두 칸 $(r_x, c_x), (r_y, c_y)$의 거리는 $|r_x - r_y| + |c_x - c_y|$이고, 이 값은 격자에서 A에서 B로 가는 최단 거리와 같습니다. 따라서, BFS를 하기 전에, 모든 마네킹을 출발점으로 하는 또다른 BFS 알고리즘을 이용해 각 칸과 가장 가까운 마네킹과의 거리를 미리 구해놓을 수 있습니다. 이 과정은 $O(NM)$이 걸립니다.

위에서 한 것 처럼 각 칸과 가장 가까운 마네킹과의 거리를 미리 구했다면, 문제의 정답을 구하는 BFS도 $O(NM)$에 구할 수 있습니다.

C++

Java

Python

소스 7

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
int dx[] = {0,0,1,-1};
int dy[] = {1,-1,0,0};
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m, k;
    cin >> n >> m >> k;
    vector<vector<int>> a(n, vector<int>(m, 0));
    vector<vector<int>> mane(n, vector<int>(m, -1));
    int sx, sy;
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            cin >> a[i][j];
            if (a[i][j] == 4) {
                sx = i;
                sy = j;
            }
        }
    }
    queue<pair<int,int>> q;
    int cnt = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            if (a[i][j] == 3) {
                q.push(make_pair(i, j));
                mane[i][j] = 0;
                cnt += 1;
            }
        }
    }
    while (!q.empty()) {
        int x = q.front().first;
        int y = q.front().second;
        q.pop();
        for (int i=0; i<4; i++) {
            int nx = x+dx[i];
            int ny = y+dy[i];
            if (0 <= nx && nx < n && 0 <= ny && ny < m) {
                if (mane[nx][ny] == -1) {
                    mane[nx][ny] = mane[x][y] + 1;
                    q.push(make_pair(nx,ny));
                }
            }
        }
    }
    if (cnt == 0) {
        for (int i=0; i<n; i++) {
            for (int j=0; j<m; j++) {
```

```
                mane[i][j] = k+1;
            }
        }
    }
    vector<vector<int>> d(n, vector<int>(m, -1));
    d[sx][sy] = 0;
    q.push(make_pair(sx,sy));
    while (!q.empty()) {
        int x = q.front().first;
        int y = q.front().second;
        q.pop();
        for (int i=0; i<4; i++) {
            int nx = x+dx[i];
            int ny = y+dy[i];
            if (0 <= nx && nx < n && 0 <= ny && ny < m) {
                if (a[nx][ny] != 1 && mane[nx][ny] > k) {
                    if (d[nx][ny] == -1) {
                        d[nx][ny] = d[x][y] + 1;
                        q.push(make_pair(nx,ny));
                    }
                }
            }
        }
    }
    int ans = -1;
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            if (a[i][j] == 2) {
                if (d[i][j] != -1) {
                    if (ans == -1 || ans > d[i][j]) {
                        ans = d[i][j];
                    }
                }
            }
        }
    }
    cout << ans << endl;
    return 0;
}
```

`mane[i][j]`에는 `(i, j)`에서 가장 가까운 마네킹과의 거리를 저장합니다. 마네킹과의 거리를 구할 때는 벽이 의미없으니 이전에 방문했는지 아닌지만 검사하면 됩니다.

격자에 마네킹이 없는 경우를 조심해야 합니다. 만약, 마네킹이 없다면 `mane[i][j]`에는 -1이 들어갈 것이고, 이렇게 되면 이후 가장 가까운 마네킹과의 거리를 계산할 때 잘못된 값과 계산을 하게 되어 정답을 구할 수 없게 됩니다. 따라서, 마네킹의 수를 `cnt`에 저장했고, 없는 경우 기둥이 아닌 모든 칸을 방문할 수 있는 것이니 `k+1`을 저장해 모든 칸과 가장 가까운 마네킹과의 거리를 `k`보다 크게 만들었습니다.

이후 `4`의 위치를 큐에 넣고 BFS를 수행했고, 최종적으로 시루가 찾아갈 수 있는 의자 중 가장 체력 소모가 적은 것을 찾아 정답으로 출력했습니다.

E번 - 방사형 그래프

문제가 두 개의 부분으로 나누어져 있습니다.

1. 능력치를 배열
2. 배열한 능력치가 볼록 다각형을 만드는지 확인

능력치의 개수는 항상 8개이고, 이 값을 N이라고 하겠습니다. 능력치를 배열하는 방법은 $N! = 8! = 40\,320$개가 있습니다.

능력치를 배열했다고 가정하고, 배열한 능력치가 볼록 다각형을 만드는지 확인하는 방법을 알아보겠습니다.

방사형 그래프는 중심을 원점으로 하는 좌표 평면으로 나타낼 수 있습니다.

1번째, 5번째 꼭짓점은 y축에 있고, 좌표는 $(0, a_1)$, $(0, -a_5)$입니다. 3번째, 7번째 꼭짓점은 x축에 있고, 좌표는 $(a_3, 0)$, $(-a_7, 0)$입니다.

2번째 꼭짓점의 좌표는 어떻게 구할 수 있을까요? 2번째 꼭짓점은 x축과 45° 각도를 이루고 있습니다. 원점과 떨어진 거리는 a_2입니다. x좌표는 cos 함수를 이용해 구할 수 있습니다. $\cos 45^\circ = x / a_2$이고, $x = \cos 45^\circ \times a_2$가 됩니다. y좌표는 sin함수를 이용해 구할 수 있습니다. $\sin 45^\circ = y / a_2$이고, $y = \sin 45^\circ \times a_2$ 입니다.

모든 꼭짓점의 좌표를 구해보면 다음과 같습니다.

1. $(0, a_1)$
2. $(\cos 45^\circ \times a_2, \sin 45^\circ \times a_2)$
3. $(a_3, 0)$
4. $(\cos 45^\circ \times a_4, -\sin 45^\circ \times a_4)$
5. $(0, -a_5)$
6. $(-\cos 45^\circ \times a_6, -\sin 45^\circ \times a_6)$
7. $(-a_7, 0)$
8. $(-\cos 45^\circ \times a_8, \sin 45^\circ \times a_8)$

모든 꼭짓점의 좌표를 구했으면, 이제 볼록 다각형인지 확인해야 합니다. 볼록 다각형은 연속하는 세 꼭짓점이 시계 방향을 이루어야 합니다. 시계 방향을 이루고 있는지 알아보기 위해 CCW를 이용할 수 있습니다.

능력치를 배열하는 방법의 수는 $O(N!)$가지이고, 배열할 때마다 볼록 다각형인지 검사하는 과정은 $O(N)$입니다. 총 $O(N! \times N)$에 해결할 수 있습니다.

소스 8

C++

Java

Python

```cpp
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;
int ccw(pair<double, double> p1, pair<double, double> p2, pair<double, double> p3) {
    double x1 = p1.first;
    double y1 = p1.second;
    double x2 = p2.first;
    double y2 = p2.second;
    double x3 = p3.first;
    double y3 = p3.second;
    double temp = x1*y2+x2*y3+x3*y1;
    temp = temp - y1*x2-y2*x3-y3*x1;
    if (temp > 0) return 1;
    else if (temp < 0) return -1;
    else return 0;
}
bool check(vector<int> &a) {
    int n = a.size();
    vector<pair<double, double>> d(n);
    double cos45 = cos(45.0 * M_PI / 180.0);
    d[0].first = 0;
    d[0].second = a[0];
    d[1].first = cos45 * a[1];
```

```
    d[1].second = cos45 * a[1];
    d[2].first = a[2];
    d[2].second = 0;
    d[3].first = cos45 * a[3];
    d[3].second = -cos45 * a[3];
    d[4].first = 0;
    d[4].second = -a[4];
    d[5].first = -cos45 * a[5];
    d[5].second = -cos45 * a[5];
    d[6].first = -a[6];
    d[6].second = 0;
    d[7].first = -cos45 * a[7];
    d[7].second = cos45 * a[7];
    for (int i=0; i<n; i++) {
        if (ccw(d[i], d[(i+1)%n], d[(i+2)%n]) != -1) {
            return false;
        }
    }
    return true;
}
int main() {
    int n = 8;
    vector<int> a(n);
    vector<int> order(n);
    for (int i=0; i<n; i++) {
        cin >> a[i];
        order[i] = i;
    }
    int ans = 0;
    do {
        vector<int> b(n);
        for (int i=0; i<n; i++) {
            b[i] = a[order[i]];
        }
        if (check(b)) {
            ans += 1;
        }
    } while(next_permutation(order.begin(), order.end()));
    cout << ans << endl;
}

```

$\cos 45^\circ$와 $\sin 45^\circ$ 의 값은 같기 때문에, 소스 코드에서는 $\cos 45^\circ$를 `cos45`에 저장하고, 두 경우 모두 이 변수를 이용했습니다.

모든 꼭짓점의 좌표는 $a + b\sqrt{2}$의 형태를 가지기 때문에, 이를 관리하는 클래스를 만들면 실수 자료형을 사용하지 않고도 소스 9와 같이 해결할 수 있습니다.

소스 9

C++

```
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;
using ll = long long;

struct Data{
    ll a, b; // a + b sqrt 2
    Data() : Data(0, 0) {}
```

```
    Data(ll a, ll b) : a(a), b(b) {}
    Data(const Data &t) : Data(t.a, t.b) {}
    Data& operator += (const Data t) {
        a += t.a; b += t.b;
        return *this;
    }
    Data& operator -= (const Data t) {
        a -= t.a; b -= t.b;
        return *this;
    }
    Data& operator *= (const Data t) {
        tie(a,b) = make_tuple(a*t.a + 2*b*t.b, a*t.b + t.a*b);
        return *this;
    }
    Data& operator *= (const ll t) {
        a *= t; b *= t;
        return *this;
    }
    Data operator + (const Data t) const { return Data(*this) += t; }
    Data operator - (const Data t) const { return Data(*this) -= t; }
    Data operator * (const Data t) const { return Data(*this) *= t; }
    Data operator * (const ll t) const { return Data(*this) *= t; }
};

using Point = pair<Data, Data>;
int Sign(ll v){ return (v > 0) - (v < 0); }
int Sign(Data t){
    if(t.a == 0 && t.b == 0) return 0;
    if(t.a >= 0 && t.b >= 0) return +1;
    if(t.a <= 0 && t.b <= 0) return -1;
    if(t.a == 0) return Sign(t.b);
    if(t.b == 0) return Sign(t.a);
    ll v = t.a*t.a - 2*t.b*t.b;
    if(t.a > 0) return Sign(v);
    return Sign(-v);
}

int CCW(const Point &p1, const Point &p2, const Point &p3){
    Data res = (p2.x - p1.x) * (p3.y - p2.y) - (p3.x - p2.x) * (p2.y - p1.y);
    return Sign(res);
}

vector<Point> MakePolygon(const vector<ll> &V){
    // cos : 1, sqrt2 / 2, 0, -sqrt2 / 2,
    //      -1, -sqrt2 / 2, 0, sqrt2 / 2
    // sin : 0, sqrt2 / 2, 1, sqrt2 / 2
    //       0, -sqrt2 / 2, -1, -sqrt2 / 2
    static constexpr int cos_a[8] = { 2, 0, 0, 0, -2, 0, 0, 0 };
    static constexpr int cos_b[8] = { 0, 1, 0, -1, 0, -1, 0, 1 };
    static constexpr int sin_a[8] = { 0, 0, 2, 0, 0, 0, -2, 0 };
    static constexpr int sin_b[8] = { 0, 1, 0, 1, 0, -1, 0, -1 };

    vector<Point> R;
    for(int i=0; i<8; i++){
        Data x = Data(cos_a[i], cos_b[i]) * V[i];
        Data y = Data(sin_a[i], sin_b[i]) * V[i];
        R.emplace_back(x, y);
    }
    return R;
```

```
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    vector<ll> V(8);
    for(auto &i : V) cin >> i;
    sort(V.begin(), V.end());

    int cnt = 0;
    vector<int> O = {0, 1, 2, 3, 4, 5, 6, 7};
    do{
        bool flag = true;
        vector<ll> W(8);
        for(int i=0; i<8; i++) W[i] = V[O[i]];
        auto P = MakePolygon(W);
        for(int i=0; i<8; i++){
            int j = (i + 1) % 8, k = (i + 2) % 8;
            if(CCW(P[i], P[j], P[k]) < 0) flag = false;
        }
        cnt += flag;
    }while(next_permutation(O.begin(), O.end()));
    cout << cnt;
}
```

Baekjoon Online Judge

스타트링크