# A. Homework

由于题目只有4种答案，不妨考虑首先全选A，此时我们一定能知道哪些题一定选A，哪些一定不选。

以此类推，策略为：先全选A，然后将错误的部分改成B，然后再将错误的改成C，最后再把错误的改成D。总共花费3次query和1次submit。

```
signed main()
{
    IOS;
    int n;
    cin >> n;
    string ans(n, 'A');
    auto ask = [&](string s) -> string
    {
        cout << "? " << s << endl;
        string res;
        cin >> res;
        return res;
    };
    auto res = ask(ans);
    for(int i = 0; i < n; i ++) if(res[i] == '0') ans[i] = 'B';
    res = ask(ans);
    for(int i = 0; i < n; i ++) if(res[i] == '0') ans[i] = 'C';
    res = ask(ans);
    for(int i = 0; i < n; i ++) if(res[i] == '0') ans[i] = 'D';
    cout << "! " << ans << endl;
    return 0;
}
```

# B. Boxes

首先，我们考虑将箱子分成两类：

- 升级后容量不变或者变大的
- 升级后容量变小的

根据贪心的思路，我们肯定先考虑扩增箱子的总容量，因此考虑先升级第一类，那第一类的箱子如何考虑升级顺序？

对于两个箱子 $(a_x, b_x), (a_y, b_y)$，钦定 $a_x \le a_y$：

- 若 $b_x \ge a_y$。则按照 $x \to y$ 的顺序，将带来 $a_x$ 的花费；而按照 $y \to x$ 的顺序，将带来 $a_y$ 的花费。所以按照 $x \to y$ 顺序更优。
- 若 $b_x < a_y$。则按照 $x \to y$ 的顺序，将带来 $a_x + a_y - b_x$ 的花费；而按照 $y \to x$ 的顺序，将带来 $a_y$ 的花费。因为 $a_1 \le b_1$，所以按照 $x \to y$ 顺序更优。

因此，考虑按 $a_i$ 从小到大模拟更优。

对于第二类箱子，考虑相同的证明：

对于两个箱子 $(a_x, b_x), (a_y, b_y)$，钦定 $b_x \ge b_y$：

- 如果 $b_x \ge a_y$。则按照 $x \to y$ 的顺序，将带来 $a_x$ 的花费；而按照 $y \to x$ 的顺序，将带来 $a_y + a_x - b_y$ 的花费。因为 $b_y - a_y < 0$，所以按照 $x \to y$ 顺序更优。
- 如果 $b_x < a_y$。则按照 $x \to y$ 的顺序，将带来 $a_x + a_y - b_x$ 的花费；而按照 $y \to x$ 的顺序，将带来 $a_y + a_x - b_y$。因为 $b_y < b_x$，所以按照 $x \to y$ 顺序更优。

因此，考虑按$b_i$从大到小模拟更优。

```
signed main()
{
    IOS;
    int n;
    cin >> n;
    vector<pii> a, b; // 表示第一类和第二类
    for(int i = 1; i <= n; i ++)
    {
        int x, y;
        cin >> x >> y;
        if(y > x) a.push_back({x, y});
        else b.push_back({x, y});
    }
    i64 ans = 0;
    // 第二类按b降序排序
    sort(all(b), [&](auto p1, auto p2) {return p1.second > p2.second;});
    // 第一类按a升序排序
    sort(all(a));
    i64 now = 0;
    for(auto[x, y] : a)
    {
        if(now < x)
            ans += x - now, now = x;
        now += y - x;
    }
    for(auto[x, y] : b)
    {
        if(now < x)
            ans += x - now, now = x;
        now -= x - y;
    }
    cout << ans << '\n';
    return 0;
}
```

# C. Keyboarding

看到数据范围 $n, m \leq 50$ ，不妨考虑bfs。

因为光标总是跳到下一个在一方向上与当前位置不同的字符，所以我们考虑预处理出每个位置上下左右能跳到哪里，方便搜索，该过程复杂度为 $O(n^3)$。

搜索中我们记录搜到的位置、已经打印出的字符数量和按键次数。

对于每次搜索，如果现在搜到的位置可以打印出字符，那么我们就将其打印出来，再向 4 个方向搜索下面的字符。注意到我们用的是广度优先搜索，所以第一次打印完所有字符一定是最优解。

**剪枝**

1. 考虑如果一个位置的一个方向上没有与这个位置不同的字符，那我们就不去搜索这个方向，可以通过预处理完成。
2. 考虑到搜索时有可能出现从一个点开始向左搜索但是没有搜到答案又向右搜回这个点的情况，我们就标记搜到一个位置最多可以打印出的字符，记作 $vis_{i,j}$ ，如果搜到一个点现在打印出的字符比 $vis_{i,j}$ 小，则剪枝。

```cpp
#include <bits/stdc++.h>
using namespace std;
int n, m, dx[100][100][5], dy[100][100][5], vis[100][100];
string a[100];
string st;
void init()
{
    for (int x = 1; x <= n; x++)
        for (int y = 1; y <= m; y++)
        {
            dx[x][y][2] = dx[x][y][4] = x, dy[x][y][1] = dy[x][y][3] = y;
            for (int i = x; i && a[x][y] == a[i][y]; i--)
                dx[x][y][1] = i;
            for (int i = x; i <= n && a[x][y] == a[i][y]; i++)
                dx[x][y][3] = i;
            for (int i = y; i <= m && a[x][y] == a[x][i]; i++)
                dy[x][y][2] = i;
            for (int i = y; i && a[x][y] == a[x][i]; i--)
                dy[x][y][4] = i;
            dx[x][y][1]--, dy[x][y][2]++, dx[x][y][3]++, dy[x][y][4]--;
        }
}
struct node
{
    int x, y, pos, cost;
};
node tmp;
queue<node> q;
bool inrange(const int &x, const int &y, const int &i)
{
    return (dx[x][y][i] >= 1 && dx[x][y][i] <= n && dy[x][y][i] >= 1 && dy[x][y][i] <= m);
}
int main()
{
    cin.tie(0) -> sync_with_stdio(0);
    cin >> n >> m;
    memset(vis, -1, sizeof(vis));
    for (int i = 1; i <= n; i++)
    {
        cin >> a[i];
        a[i] = '#' + a[i];
    }
    cin >> st;
    st = st + "*";
    init();
    q.push({1, 1, 0, 0});
    while (!q.empty())
    {
        tmp = q.front();
        q.pop();
        if (a[tmp.x][tmp.y] == st[tmp.pos])
        {
            if (tmp.pos == st.size() - 1)
            {
                cout << tmp.cost + 1 << "\n";
                return 0;
            }
            vis[tmp.x][tmp.y] = tmp.pos + 1, q.push({tmp.x, tmp.y, tmp.pos + 1, tmp.cost + 1});
        }
        else
        {
            for (int i = 1; i <= 4; i++)
                if (inrange(tmp.x, tmp.y, i) && vis[dx[tmp.x][tmp.y][i]][dy[tmp.x][tmp.y][i]] < tmp.pos)
                    vis[dx[tmp.x][tmp.y][i]][dy[tmp.x][tmp.y][i]] = tmp.pos,
                    q.push({dx[tmp.x][tmp.y][i], dy[tmp.x][tmp.y][i], tmp.pos, tmp.cost + 1});
        }
    }
```

```
    return 0;
}
```

# D. Internet of Everything

这里给出2种做法：

## 方法1：并查集

操作 1是单点修改，操作2是对某个集合修改，所以我们希望找到一种可以对整个集合进行统一操作的数据结构，考虑并查集。

先考虑操作 2，假设有集合 $s$，根为 $k$ ，值为 $x$ ，那么我们直接将 $k$ 的父亲置为值为 $x+1$ 的集合的根即可，我们发现至少需要建立 $2$ 个映射关系：

1. 值到根的映射，代码中用 $idx[x]$ 表示值为 $x$ 的集合的根。
2. 根到值的映射，代码中用 $val[k]$ 表示根为 $k$ 的集合的值。

那么操作 2 就可以表示为：

$$s[idx[x]] = idx[x+1]$$

同理操作 1 可以表示为：

$$s[x] = idx[val[find(x)]+1]$$

但是会出现一个问题，就是如果后续有值为 $x-1$ 的集合 $s'$ 进行了 2 操作，就会导致 $s'$ 指向 $s$ ，而现在的 $s$ 集合实际上已经指向值为 $x+1$ 的集合了，所以 $s'$ 最终指向的其实是值为 $x+1$ 的集合，就会出现问题。

本质原因其实是因为现在已经没有代表值 $x$ 的根了，所以我们可以考虑新建一个并查集节点作为代表值 $x$ 的根，同时令原先的根无效化，代码中用 $extend$ 函数实现。

还有一个注意点，即所有集合的根必须从 $n+1$ 开始，包括后面 $extend$ 出来的新根，为什么呢，假设我们将第 $1$ 个节点作为值 $0$ 的根，第 $2$ 个节点作为值 $1$ 的根，那么一开始所有的节点值都为 $0$ ，所以所有节点包括节点 $1$ 本身都会指向节点 $1$ ，这个时候如果对节点 $1$ 进行操作 $1$ ，节点 $1$ 的值从 $0$ 变为 $1$ ，从而指向节点 $2$ ，这个时候节点 1 和节点 2 就会成为 "共轭父子" ，导致 find 函数失效，所以一开始代表值 $0$ 的根为 $n+1$ ，$q$ 次操作值至多为 $q$ ，一开始利用 $extend$ 函数先建出 $q+1$ 个根，索引从 $n+1$ 到 $n+q+1$ ，后续需要建立新根时依次递推。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define IOS ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
// #define int long long
#define endl '\n'
#define all(_x) _x.begin(), _x.end()
#define range(_x, _st, _ed) (_x.begin() + _st), (_x.begin() + _ed)
#define rep(_x, _y, _z) for (int _x = _y; _x <= _z; _x++)
#define matrix(_x, _y, _z) vector<vector<int>>(_x, vector<int>(_y, _z))
#define debug(_x) cout << #_x << '=' << _x << endl
using i64 = long long;
using i128 = __int128;
using pii = pair<int, int>;
using mat = vector<vector<int>>;
using u64 = unsigned long long;
constexpr int N = 2e5 + 10;
// dont use umap!!!

void solve()
{
    int n, q;
    cin >> n >> q;
    vector<int> f(n + 1, n + 1), idx2val(n + 1, 0), val2idx(q + 1, -1);
    auto find = [&](auto &self,int x) -> int
    {
        return f[x] == x ? x : f[x] = self(self, f[x]);
    };
    auto create = [&](int x)
    {
        val2idx[x] = f.size();
        f.push_back(f.size());
        idx2val.push_back(x);
    };
    create(0);
    while(q--)
    {
        int op, x;
        cin >> op >> x;
        if(op == 1)
        {
            if(val2idx[idx2val[find(find, x)] + 1] == -1)
                create(idx2val[find(find, x)] + 1);
            f[x] = val2idx[idx2val[find(find, x)] + 1];
        }
        else
        {
            if(val2idx[x + 1] == -1) create(x + 1);
            f[val2idx[x]] = val2idx[x + 1];
            create(x);
        }
    }
    for(int i = 1; i <= n; i ++)
    {
        cout << idx2val[find(find, i)] << " \n"[i == n];
    }
}

signed main()
{
    IOS;
    int _ = 1;
    cin >> _;

    while (_--) solve();

    return 0;
}
```

## 方法2:启发式合并

考虑对整个值域建立 set 的数组，set 中存储该值对应的数组下标，维护数组 sval、posval、posidx 分别代表每个集合维护的值、每个值对应的集合、每个下标对应的集合，合并 set 时优先将小的合并到大的。由于平均每次合并的复杂度为 $O(\log n)$，而维护以上数组并不需要额外复杂度，因此最终复杂度为$O(q \log n)$。

```
void solve(int n, int q)
{
    int m = max(n, q) + 1;
    vector<set<int>> s(m);
    vector<int> sval(m);
    vector<int> posval(m), posidx(m, 0);

    iota(sval.begin(), sval.end(), 0);
    iota(posval.begin(), posval.end(), 0);
    for (int i = 1; i <= n; i++)
        s[0].insert(i);

    while (q--)
    {
        int op, x;
        cin >> op >> x;
        if (op == 1)
        {
            s[posidx[x]].erase(x);
            posidx[x] = posval[ sval[posidx[x]] + 1 ];
            s[posidx[x]].insert(x);
            continue;
        }

        if (s[posval[x]].size() <= s[posval[x + 1]].size())
        {
            for (int u : s[posval[x]])
            {
                s[posval[x + 1]].insert(u);
                posidx[u] = posval[x + 1];
            }
            s[posval[x]].clear();
        }
        else
        {
            for (int u : s[posval[x + 1]])
            {
                s[posval[x]].insert(u);
                posidx[u] = posval[x];
            }
            s[posval[x + 1]].clear();
            swap(posval[x], posval[x + 1]);
            swap(sval[posval[x]], sval[posval[x + 1]]);
        }
    }

    vector<int> ans(n + 1);
    for (int i = 0; i < m; i++)
    {
        for (int u : s[i])
            ans[u] = sval[i];
    }
    for (int i = 1; i <= n; i++)
        cout << ans[i] << ' ';
    cout << endl;
}
```