第十届校程序设计竞赛题解

> 出题：牟其霖
>
> 数据：牟其霖、倪翊诚
>
> 验题：USTS-ACM集训队现役成员
>
> 题目数据和spj代码等相关资料已公开至 Github

Problem A Codeforces

算法标签：无

直接输出满足题意要求的a,b,c即可

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    cout << "3 4 2" << endl;
    return 0;
}
```

时间复杂度：$O(1)$

Problem B 回文字符串

算法标签：字符串

对于长度为 n 的被修改过一个字符的回文字符串，遍历该字符串前 $\lfloor \frac{n}{2} \rfloor$ 个字符，如果第 i 个字符和第 $n-i+1$ 个字符不相同，那么两者其一一定是被修改过的一个字符，题目要求输出下标小的，直接输出 i 即可

特殊地，对于长度n为奇数的字符串，恰好修改的是第 $\lfloor \frac{n}{2} \rfloor + 1$个字符，那么修改后的字符串仍然是回文的，在这种情况下，唯一可能被修改的就是第 $\lfloor \frac{n}{2} \rfloor + 1$个字符，输出 $n/2+1$ 即可

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    string s;
    cin >> s;
    // 遍历前n/2个字符，检查对称字符是否相同
    for (int i = 0; i < n / 2; i++) {
        if (s[i] != s[n - i - 1]) {
            // 找到了被修改的位置，输出下标
            cout << i + 1 << endl;
            // 直接退出程序
```

```
            return 0;
        }
    }
    // 程序走到了这里，说明字符串前n/2个字符满足回文且字符串长度为奇数
    // 被修改的位置只能是n/2+1
    cout << n / 2 + 1 << endl;
    return 0;
}
```

时间复杂度：$O(n)$

Problem C 叠被子

算法标签：数学

问题等价于，对于被子的每个整数位置，两端是否有同时有偶数个线段，记录这样的整数位置的数量。

因此可以写一个O(n)的算法：

```
int ans = 0;
// 对于长度为n的被子，有n+1个整数位置：0~n
for (int i = 0; i <= n; i++) {
    // 判断左右线段数量是否同时为偶数
    if (i % 2 == 0 && (n - i) % 2 == 0) {
        ans++;
    }
}
cout << ans << endl;
```

但是O(n)的算法在本题n为1e9的情况下是会超时的

我们可以发现，对于奇数长度的被子，无论选择哪一个位置，肯定会有一端的线段数量为奇数，而这种叠法不合法，因此奇数长度的被子答案永远为0

对于偶数长度的被子，如果被选择的位置为奇数，那么会导致左右两端都是奇数个线段，也不合法；如果被选择的位置为偶数，则左右两端线段树也一定为偶数且合法。因此对于偶数长度n的被子的答案，我们只需要输出$0-n$范围内有多少个偶数即可，直接O(1)输出答案

参考代码

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    // 奇数，0
    if (n & 1) cout << 0 << endl;
    // 偶数，0~n范围内一共有1+n/2个偶数
    else cout << n / 2 + 1 << endl;

    return 0;
}
```

时间复杂度：$O(1)$

Problem D 双拼与全拼

算法标签：模拟

开两个变量：double_cnt 和 full_cnt，记录每句话需要打几个字母，越少的一方打字更快

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        int len;
        cin >> len;
        int double_cnt = 0;
        int full_cnt = 0;
        for (int j = 1; j <= len; j++) {
            int a;
            cin >> a;
            // 无论a为多少，双拼总是需要打两个字母
            double_cnt += 2;
            full_cnt += a;
        }
        if (double_cnt > full_cnt) cout << "Full" << endl;
        else if (full_cnt > double_cnt) cout << "Double" << endl;
        else cout << "Both" << endl;
    }
    return 0;
}
```

时间复杂度：$O(\sum_{i=1}^{n} len_i)$

Problem E 构造数列

算法标签：构造，贪心

首先，无论 n 和 len 是怎么样的，对于前1~len个字符，我们可以先确定下来，前 len 个数字我们不妨这样构造：

`0 1 2 ... len-1`

例如，当 $n=6$，$len=3$ 时，我们先构造前3个数字：`0 1 2`，此时前3个数字的MEX值：
$MEX(\{0,1,2\})=3=len$ 符合题意

现在，窗口向后移动一格，我们假设新加进来的数字是`?`，那么现在情况是这样的：`0 {1 2 ?}`，其中大括号为窗口。我们现在需要让 $MEX(\{1,2,?\})=3=len$，此时发现，只有当`?`为0的时候，才能符合题意。同时我们发现，`0` 就是在窗口向后移动时，被去掉的那个数字，此时我们需要把去掉的数字补回来即可

接着，窗口再向后移动一格，同样的，我们需要让此时的`?`为1才能符合题意，再移动一格同理

按照上述方法最后构造出来的数列最终为`0 1 2 0 1 2`，此时可以发现规律，数列在进行一个长度为 len 的循环，合法性在上述已证明

最后呈现的代码及其简洁，即做一个长度为 len 的循环

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    int n,len;
    cin >> n >> len;
    for (int i = 0; i < n; i++) {
        cout << i % len << ' ';
    }
    return 0;
}
```

时间复杂度：$O(n)$

Problem F 挖隧道

算法标签：前缀和，差分

如果对于每次询问，我们都遍历一次泥土山的每一列，那么时间复杂度是$O(nq)$，显然会超时，因此我们需要使用算法来进行优化。在理解接下来的内容前，请先理解前缀和与差分算法

- 对于泥土山的第i列，有a_i个泥土，那么在竖直方向上看，第1行到第a_i行的每一各均存在一个泥土，因此我们可以使用差分数组，记录竖直方向上的泥土数并在枚举完所有的列后对差分数组进行累加
- 在差分数组累加完后，我们得到的是每一行的泥土数，如果要问第1行到第h行的泥土总数，我们依然枚举第1行到第h行，因此我们可以使用前缀和，再从下到上累加泥土数量，每次询问可以直接O(1)得到答案。**注意这里前缀和数组的上限为泥土山最大高度的上限，我们直接使用题目给定的上限即可**
- 注意数据范围，最大的情况下我们的数据会超过10^{10}，因此C++和Java选手不能使用int变量

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

// 最大高度
const int MAX_H = 100010;

int main() {
    int n,q;
    cin >> n >> q;
    // 开long long
    vector<long long> c(MAX_H);
    for (int i = 1; i <= n; i++) {
        int a;
        cin >> a;
        // 差分
        c[1]++;
        c[a+1]--;
    }
    // 差分累加

```

```
    for (int i = 1; i <= MAX_H; i++) {
        c[i] += c[i-1];
    }
    // 前缀和累加
    for (int i = 1; i <= MAX_H; i++) {
        c[i] += c[i-1];
    }
    while (q--) {
        int h;
        cin >> h;
        // 直接输出答案，这里完整的应该是c[h] - c[0]
        cout << c[h] << ' ';
    }
    return 0;
}
```

时间复杂度：$O(n+q+\max_{i=1}^{n} a_i)$

Problem G 二进制月

算法标签：DFS、二进制、位运算 | 图论、BFS

首先我们要明确题意：

- 选中的数字相加后不能进位，即选中的数字相加后，每一位存在且只能存在一个1

要快速解决上述问题，我们对于十进制的两个数字，直接使用"与"操作即可判断相加后是否进位。同时对于两个一模一样的数字，我们只能使用一个，所以我们需要对数字进行去重

在编写代码的过程中，我们还需要对字符串和数字进行双向的转化。对于这题，在C++中我们可以使用`bitset<5>(a)`快速得到a的二进制表示

在上述的基础上，我们进行DFS爆搜即可

时间复杂度分析

我们直接粗略的来算一下时间复杂度：去重后最多有32个数，那么在DFS的时候，对于每个数我们有选和不选两种操作，因此乍一看复杂度为$O(2^{32})$。但是这显然是不对。我们在DFS一定是会有剪枝的操作：使用"与"操作判断相加后是否进位。所以在DFS时，我们最多选5个数，即`10000`,`01000`，`01000`，`01000`，`00001`（当然`00000`也可以算进去），所以我们最终的时间复杂度应该取最大情况的上限，即$O(2^5)$

其他做法

可以考虑去重后对所有数建一个完全无向图，从值为0开始BFS，每次寻找不进位的数字相加并且距离+1，过程中搜到答案为31即可输出距离和路径

参考代码

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    // 使用set去重
    set<string> s;
```

```
    for (int i = 1; i <= n; i++) {
        string t;
        cin >> t;
        s.insert(t);
    }
    // 把去重后的数字转化为十进制存储到a中
    vector<int> a;
    for (auto c : s) {
        // 2^4
        int cnt = 16;
        // 临时变量，存当前十进制的值
        int t = 0;
        // 从最高位向最低位枚举
        for (int i = 0; i < 5; i++) {
            // 如果是1就加上cnt
            if (c[i] == '1') t += cnt;
            // cnt右移一位
            cnt >>= 1;
        }
        // 讲十进制的值存入a中
        a.push_back(t);
    }
    // 将n更新为去重后的数组大小
    n = a.size();
    // dfs时记录当前数是否被用过
    vector<int> vis(n,0);
    // 答案数组，记录答案，使用bitset方便存储
    vector<bitset<5>> ans;
    // 是否有答案
    int flag = 0;
    function<void(int)> dfs = [&] (int cur) {
        // 已经有答案了，结束dfs
        if (flag) return;
        // 当前数正好为11111即31，记录答案
        if (cur == 31) {
            // 输出答案
            cout << ans.size() << '\n';
            for (auto b : ans) {
                cout << b << '\n';
            }
            // 记录已经有答案了
            flag = 1;
            return;
        }
        // 遍历所有没有被使用过的数字
        for (int i = 0; i < n; i++) {
            // 被使用了
            if (vis[i]) continue;
            // 产生进位了
            if (cur & a[i]) continue;
            // 放到答案里
            ans.push_back(bitset<5>(a[i]));
            // 记录被使用了
            vis[i] = 1;
            // 继续dfs，将他们相加，二进制对应为”或“操作
            dfs(cur | a[i]);
            // dfs的回溯
```

```
            ans.pop_back();
            vis[i] = 0;
        }
    };
    // 从0开始dfs
    dfs(0);
    // 如果没有答案，输出-1
    if (!flag) cout << "-1";
    return 0;
}
```

时间复杂度：$O(4^{|a|})$

Problem H Slim走迷宫

算法标签：搜索、二分、二维前缀和

我们需要使用三种算法来完成这道题：

- 二分：验证当slim为$r \times r$的大小时，是否可以走到终点
- 搜索：模拟slim大小为 $r \times r$ 时，在迷宫内移动
- 二维前缀和：判断slim在移动时是否会碰到障碍，能否往指定方向前进

以下展示使用**BFS**完成搜索（亦可使用DFS）

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;

// 四个方向的枚举
int dx[] = {-1,0,1,0}, dy[] = {0,1,0,-1};

int main() {
    int n,m;
    cin >> n >> m;
    // g为迷宫地图，c为二维前缀和数组
    vector<vector<int>> g(n+2,vector<int>(m+2)), c(n+2,vector<int>(m+2));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> g[i][j];
            // 二维前缀和累加操作
            c[i][j] = g[i][j] + c[i-1][j] + c[i][j-1] - c[i-1][j-1];
        }
    }

    // bfs模拟边长为r的slim走迷宫
    auto bfs = [&] (int r) {
        // 定义bfs队列
        queue<pair<int,int>> q;
        // 定义当前点是否已经经过
        vector<vector<int>> vis(n+2,vector<int>(m+2,0));
        // 从起点开始bfs走迷宫
        q.push({r,r});
        // 只要队列不空
```

```
        while (q.size()) {
            // 取出队头
            auto t = q.front();
            q.pop();
            int x = t.first, y = t.second;
            // 如果经过了就不用继续了
            if (vis[x][y]) continue;
            // 标记经过
            vis[x][y] = 1;
            // 使用二维前缀和计算当前slim内障碍数量
            int cnt = c[x][y] - c[x][y-r] - c[x-r][y] + c[x-r][y-r];
            // 如果没有障碍，且到达了终点，那么边长为r时可以通过
            if (cnt == 0 && x == n && y == m) return true;
            // 如果有障碍，则当前位置不合法
            if (cnt) continue;
            // 枚举四个方向
            for (int i = 0; i < 4; i++) {
                // 下一个点的坐标(a,b)
                int a = x + dx[i], b = y + dy[i];
                // 是否越界，是否已经经过
                if (a - r < 0 || a > n || b - r < 0 || b > m || vis[a][b]) continue;
                // 加入到队列中
                q.push({a,b});
            }
        }
        // 到这里，说明没有走到终点
        return false;
    };

    // 二分答案
    int l = 1, r = min(n,m);
    while (l < r) {
        int mid = l + r + 1 >> 1;
        if (bfs(mid)) l = mid;
        else r = mid - 1;
    }
    cout << l << endl;
    return 0;
}
```

时间复杂度：$O(nm\log(\min(n,m))$