> 主要受翟博士影响，基本全部改版，没有原题出现，前两道送分题，后三道出自 CCF 题库，能度很高，所以很多做哭了基本上，所以能刷的就刷吧，但是历年的 题也要好好敲敲的，也是有可能出原题的。相信自己，进复试的基本水平差不多。 难大家都难，只是心疼软工专硕，初试基本很少敲代码，纯理论题，上机基本靠 历年真题存活，结果.....唉.....

1. **跳一跳(15 分)**

   近来，跳一跳这款小游戏风靡全国，受到不少玩家的喜爱。

   简化后的跳一跳规则如下：玩家每次从当前方块跳到下一个方块，如果没有跳到 下一个方块上则游戏结束。 如果跳到了方块上，但没有跳到方块的中心则获得 1 分；跳到方块中心时，若上 一次的得分为 1 分或这是本局游戏的第一次跳跃则此次得分为 2 分，否则此次 得分比上一次得分多两分（即连续跳到方块中心时，总得分将+2，+4，+6，+8...）。

   现在给出一个人跳一跳的全过程，请你求出他本局游戏的得分（按照题目描述的 规则）。

   输入包含多个数字，用空格分隔，每个数字都是 1，2，0 之一，1 表示此次跳跃 跳到了方块上但是没有跳到中心，2 表示此次跳跃跳到了方块上并且跳到了方块 中心，0 表示此次跳跃没有跳到方块上（此时游戏结束）。

   输出一个整数，为本局游戏的得分（在本题的规则下）。

   输入样例：1 1 2 2 2 1 1 2 2 0

   输出样例：22

   数据规模和约定 对于所有评测用例，输入的数字不超过 30 个，保证 0 正好出现一次且为最后一个数字

```c
#include<stdio.h>
int main(){
    // 用于存储前一个数字的值，初始化为 0
    int pre = 0, cur;
    // 存储最终的结果
    int ans = 0;
    // 当输入的数字不为 0 时，继续执行循环
    while (~scanf("%d", &cur) && cur) {
        if (cur == 1) {
            ans += 1;
            pre = 1;
        }
        else if (pre == 1 || pre == 0) {
            ans += 2;
            pre = 2;
        }
        else {
            ans += pre + 2;
            // 前一个数字更新为前一个数字的值加 2
            pre = pre + 2;
        }
    }
    // 输出最终结果
    printf("%d\n", ans);
    return 0;
}
```

2. **计算日期（25 分）**

   问题描述

   给定一个年份 y 和一个整数 d，问这一年的第 d 天是几月几日？ 注意闰年的 2 月有 29 天。

   满足下面条件之一的是闰年：

   1）年份是 4 的整数倍，而且不是 100 的整数倍；

   2）年份是 400 的整数倍。

   输入格式

   - 输入的第一行包含一个整数 y，表示年份，年份在 1900 到 2015 之间（包含 1900 和 2015）。
   - 输入的第二行包含一个整数 d，d 在 1 至 365 之间。

输出格式

- 输出两行，每行一个整数，分别表示答案的月份和日期。

样例输入 2015 80

样例输出 3 21

样例输入 2000 40

样例输出 2

参考代码：

```
#include<iostream>
using namespace std;
// 函数用于判断是否为闰年
bool isleapyear(int year)
{
    // 如果能被 400 整除或者能被 4 整除但不能被 100 整除，则为闰年
    if (year % 400 == 0 || (year % 4 == 0 && year % 100!= 0))
        return true;
    return false;
}
int main()
{
    // 存储输入的年份
    int year;
    // 存储输入的天数
    int d;
    // 从标准输入读取年份和天数
    cin >> year >> d;
    // 存储每个月的天数，默认 2 月为 28 天
    int months[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    // 如果是闰年，将 2 月的天数修改为 29 天
    if (isleapyear(year))
        months[1] = 29;
    // 从 0 开始表示第一个月
    int month(0);
    // 当输入的天数大于当前月的天数时，进入循环
    while (d > months[month])
    {
        // 从输入的天数中减去当前月的天数
        d -= months[month];
        // 进入下一个月
        month++;
    }
    // 输出当前所在的月份（从 1 开始计数）
    cout << month + 1 << endl;
    // 输出当前月的第几天
    cout << d << endl;
    return 0;
}
```

3. **交通规划（30 分）**

G 国国王来中国参观后，被中国的高速铁路深深的震撼，决定为自己的国家也建 设一个高速铁路系统。 建设高速铁路投入非常大，为了节约建设成本，G 国国王决定不新建铁路， 而是将已有的铁路改造成高速铁路。现在，**请你为 G 国国王提供一个方案，将现 有的一部分铁路改造成高速铁路，使得任何两个城市间都可以通过高速铁路到达， 而且从所有城市乘坐高速铁路到首都的最短路程和原来一样长。**请你告诉 G 国 国王在这些条件下最少要改造多长的铁路。

输入的第一行包含两个整数 n, m，分别表示 G 国城市的数量和城市间铁路 的数量。所有的城市由 1 到 n 编号，首都为 1 号。 接下来 m 行，每行三个 整数 a, b, c，表示城市 a 和城市 b 之间有一条长度为 c 的双向铁路。这条铁路不会经过 a 和 b 以外的城市。

输出一行，表示在满足条件的情况下最少要改造的铁路长度。

Input

```
4 5
1 2 4
1 3 5
2 3 2
2 4 3
3 4 2
```

Output

```
11
```

评测用例规模与约定

对于 20%的评测用例，1<=n<=10,1<=m<=50;

对于 50%的评测用例，1<=n<=100,1<=m<=5000;

对于 80%的评测用例，1<=n<=1000,1<=m<=50000;

对于 100%的评测用例，1<=n<=10000,1<=m<=100000;1<=a,b<=n,1<=c<=10 00,输入保证每个城市都可以通过铁路达到首都

参考代码：

```cpp
#include<cstdio>
#include<iostream>
#include<sstream>
#include<cstdlib>
#include<cstring>
#include<string>
#include<climits>
#include<cmath>
#include<algorithm>
#include<queue>
#include<vector>
#include<stack>
#include<set>
#include<cctype>
#include<map>
#include<utility>
using namespace std;
// 定义一个宏，用于打印变量及其名称
#define print(A) cout << #A << ": "<< A << endl;
// 定义一个很大的数表示无穷大
#define inf 0x3fffffff
// 定义一个长整型别名
typedef long long int ll;

// 定义边结构体，包含边的起点、终点和距离
struct edge {
    int head, tail, dist;
    edge(int h = -1, int t = -1, int d = -1): head(h), tail(t), dist(d) {}
};

// 定义堆节点结构体，包含节点编号和到源节点的距离
struct headnode {
    int node, dist;
    headnode(int n = -1, int d = -1): node(n), dist(d) {}
    // 重载小于运算符，用于优先队列的比较，距离小的优先
    bool operator<(const headnode& another) const { return dist > another.dist; }
};

int main(int argc, char const *argv[]) {
```

```
    // 输入节点数和边数
    int node_num, edge_num; cin >> node_num >> edge_num;
    // 存储每个节点到源节点的最短距离，初始化为无穷大
    vector<int> d;
    d.resize(node_num+1, inf);
    // 标记节点是否已经处理过，初始化为 false
    vector<bool> done;
    done.resize(node_num+1, false);
    // 存储图的邻接表
    vector<vector<int> > graph;
    graph.resize(node_num+1, vector<int>());
    // 存储所有的边
    vector<edge> edges;
    edges.resize(edge_num+1);
    // 存储每个节点离上一节点的距离
    vector<int> last_add;
    last_add.resize(node_num+1, inf);

    // 输入边的信息并构建图
    for (int i = 1; i <= edge_num; i++) {
        int node1, node2, dist;
        cin >> node1 >> node2 >> dist;
        edges[i] = edge(node1, node2, dist);
        // 将边的编号加入到节点的邻接表中
        graph[node1].push_back(i);
        graph[node2].push_back(i);
    }

    // 定义优先队列，存储堆节点
    priority_queue<headnode> pq;
    // 将源节点加入优先队列，距离为 0
    pq.push(headnode(1, 0));
    last_add[1] = 0;
    d[1] = 0;
    // 当优先队列不为空时
    while (!pq.empty()) {
        // 取出堆顶节点
        headnode top = pq.top(); pq.pop();
        // 如果该节点已经处理过，跳过
        if (done[top.node]) continue;
        // 标记该节点已处理
        done[top.node] = true;
        // 遍历该节点的邻接边
        for (int i = 0; i < graph[top.node].size(); i++) {
            int index = graph[top.node][i];
            edge temp = edges[index];
            // 找出边的另一个节点
            int another_node = (top.node == temp.head)? temp.tail: temp.head;
            // 如果通过该边到达另一个节点的距离更短
            if (d[another_node] > d[top.node]+temp.dist) {
                // 更新 last_add 的值
                last_add[another_node] = temp.dist;
                // 更新最短距离
                d[another_node] = d[top.node]+temp.dist;
                // 将另一个节点加入优先队列
                pq.push(headnode(another_node, d[another_node]));
            }
            // 如果通过该边到达另一个节点的距离相等
            else if (d[another_node] == d[top.node]+temp.dist) {
                // 更新 last_add 的最小值
                last_add[another_node] = min(temp.dist, last_add[another_node]);
```

```
            }
        }
    }
    // 计算结果
    int res = 0;
    for (int i = 1; i <= node_num; i++) res += last_add[i];
    cout << res << endl;
    return 0;
}
```

4. **最短路径（30 分）**

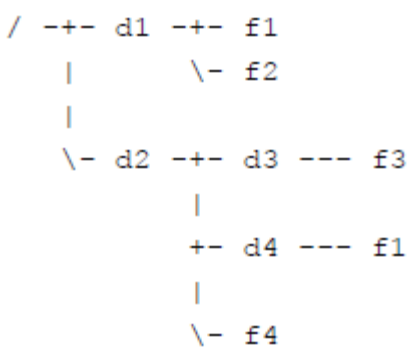
在操作系统中，数据通常以文件的形式存储在文件系统中。文件系统一般采用层 次化的组织形式，由目录（或者文件夹）和文件构成，形成一棵树的形状。文件 有内容，用于存储数据。目录是容器，可包含文件或其他目录。同一个目录下的 所有文件和目录的名字各不相同，不同目录下可以有名字相同的文件或目录。

为了指定文件系统中的某个文件，需要用路径来定位。在类 Unix 系统（Linux、Max OS X、FreeBSD 等）中，路径由若干部分构成，每个部分是一个目 录或者文件的名字，相邻两个部分之间用 / 符号分隔。

有一个特殊的目录被称为根目录，是整个文件系统形成的这棵树的根节点， 用一个单独的 / 符号表示。在操作系统中，有当前目录的概念，表示用户目前 正在工作的目录。根据出发点可以把路径分为两类：

- 绝对路径：以 / 符号开头，表示从根目录开始构建的路径。
- 相对路径：不以 / 符号开头，表示从当前目录开始构建的路径。

例如，有一个文件系统的结构如下图所示。在这个文件系统中，有根目录 / 和其他普通目录 d1、d2、d3、d4，以及文件 f1、f2、f3、f1、f4。其中，两个 f1 是同名文件，但在不同的目录下。



对于 d4 目录下的 f1 文件，可以用绝对路径 /d2/d4/f1 来指定。如果当前 目录是 /d2/d3，这个文件也可以用相对路径 ../d4/f1 来指定，这里 .. 表示上一 级目录（注意，根目录的上一级目录是它本身）。还有 . 表示本目录，例如 /d1/./f1 指定的就是 /d1/f1。注意，如果有多个连续的 / 出现，其效果等同于一个 /， 例如 /d1///f1 指定的也是 /d1/f1。

本题会给出一些路径，要求对于每个路径，给出正规化以后的形式。一个路 径经过正规化操作后，其指定的文件不变，但是会变成一个不包含 . 和 .. 的绝 对路径，且不包含连续多个 / 符号。如果一个路径以 / 结尾，那么它代表的一 定是一个目录，正规化操作要去掉结尾的 /。若这个路径代表根目录，则正规化 操作的结果是 /。若路径为空字符串，则正规化操作的结果是当前目录。

输入格式 第一行包含一个整数 P，表示需要进行正规化操作的路径个数。 第二行包含一个字符串，表示当前目录。 以下 P 行，每行包含一个字符串，表示需要进行正规化操作的路径。

输出格式 共 P 行，每行一个字符串，表示经过正规化操作后的路径，顺序与输入对 应。

样例输入

```
7
/d2/d3
/d2/d4/f1
../d4/f1
/d1/./f1
/d1///f1
/d1/
///
/d1/../../d2
```

样例输出

```
/d2/d4/f1
/d2/d4/f1
/d1/f1
/d1/f1
/d1
/
/d2
```

评测用例规模与约定 1 ≤ P ≤ 10。

- 文件和目录的名字只包含大小写字母、数字和小数点 .、减号 - 以及下划线 _。
- 不会有文件或目录的名字是 . 或 .. ，它们具有题目描述中给出的特殊含义。

输入的所有路径每个长度不超过 1000 个字符。 输入的当前目录保证是一个经过正规化操作后的路径。

参考代码：

```cpp
#include<cstdio>
#include<iostream>
#include<sstream>
#include<cstdlib>
#include<cstring>
#include<string>
#include<climits>
#include<cmath>

#include<algorithm>
#include<queue>
#include<vector>
#include<stack>
#include<set>
#include<cctype>
#include<map>
#include<utility>
using namespace std;
// 定义一个宏，用于打印变量及其名称
#define print(A) cout << #A << ": "<< A << endl;
// 定义一个很大的数表示无穷大
#define inf 0x3fffffff
// 定义一个长整型别名
typedef long long int ll;

// 定义边结构体，包含边的起点、终点和距离
struct edge {
    int head, tail, dist;
    edge(int h = -1, int t = -1, int d = -1): head(h), tail(t), dist(d) {}
};

// 定义堆节点结构体，包含节点编号和到源节点的距离
struct headnode {
    int node, dist;
    headnode(int n = -1, int d = -1): node(n), dist(d) {}
    // 重载小于运算符，用于优先队列的比较，距离小的优先
    bool operator<(const headnode& another) const { return dist > another.dist; }
};

int main(int argc, char const *argv[]) {
    // 输入节点数和边数
    int node_num, edge_num; cin >> node_num >> edge_num;
    // 存储每个节点到源节点的最短距离，初始化为无穷大
    vector<int> d;
```

```
    d.resize(node_num + 1, inf);
    // 标记节点是否已经处理过，初始化为 false
    vector<bool> done;
    done.resize(node_num + 1, false);
    // 存储图的邻接表
    vector<vector<int> > graph;
    graph.resize(node_num + 1, vector<int>());
    // 存储所有的边
    vector<edge> edges;
    edges.resize(edge_num + 1);
    // 存储每个节点离上一节点的距离
    vector<int> last_add;
    last_add.resize(node_num + 1, inf);

    // 输入边的信息并构建图
    for (int i = 1; i <= edge_num; i++) {
        int node1, node2, dist;
        cin >> node1 >> node2 >> dist;
        edges[i] = edge(node1, node2, dist);
        // 将边的编号加入到节点的邻接表中
        graph[node1].push_back(i);
        graph[node2].push_back(i);
    }

    // 定义优先队列，存储堆节点
    priority_queue<headnode> pq;
    // 将源节点加入优先队列，距离为 0
    pq.push(headnode(1, 0));
    // 将源节点的 last_add 初始化为 0，表示距离源节点最近的边的距离为 0
    last_add[1] = 0;
    // 源节点到自身的最短距离为 0
    d[1] = 0;
    // 当优先队列不为空时
    while (!pq.empty()) {
        // 取出堆顶节点
        headnode top = pq.top(); pq.pop();
        // 如果该节点已经处理过，跳过
        if (done[top.node]) continue;
        // 标记该节点已处理
        done[top.node] = true;
        // 遍历该节点的邻接边
        for (int i = 0; i < graph[top.node].size(); i++) {
            int index = graph[top.node][i];
            edge temp = edges[index];
            // 找出边的另一个节点
            int another_node = (top.node == temp.head)? temp.tail : temp.head;
            // 如果通过该边到达另一个节点的距离更短
            if (d[another_node] > d[top.node] + temp.dist) {
                // 更新 last_add 的值，存储新的最短边的距离
                last_add[another_node] = temp.dist;
                // 更新最短距离
                d[another_node] = d[top.node] + temp.dist;
                // 将另一个节点加入优先队列，更新其距离信息
                pq.push(headnode(another_node, d[another_node]));
            }
            // 如果通过该边到达另一个节点的距离相等
            else if (d[another_node] == d[top.node] + temp.dist) {
                // 更新 last_add 的最小值
                last_add[another_node] = min(temp.dist, last_add[another_node]);
            }
        }
```

```
    }
    // 计算结果
    int res = 0;
    for (int i = 1; i <= node_num; i++) res += last_add[i];
    cout << res << endl;
    return 0;
}
```