# 锦乐寻宝

如果先破解地图，再挖掘，需要 $4+5$ 天必然能挖掘出宝藏，剩余价值 $1437-90=1347$

如果锦乐不破解地图，随机选择一个地方挖掘 5 天，如果没挖到，再花 5 天时间在另一个地方挖掘。那么有 $50\%$ 概率花 5 天直接得到宝藏；有 $50\%$ 概率花 10 天直接得到宝藏，期望剩余价值 $1437-0.5\times 50-0.5\times 100=1362$

所以答案是 $1362$

# 猜一猜

对二进制位 $a,b$，进行或运算和异或运算的结果如下表所示：

| a,b | 或 | 异或 |
| --- | --- | --- |
| 0,0 | 0 | 0 |
| 0,1 | 1 | 1 |
| 1,0 | 1 | 1 |
| 1,1 | 1 | 0 |

对每个二进制位都是 $0,1$ 随机来说。如果是或运算(代码 `A`)，那么每个位应当有 $75\%$ 概率为 $1$，否则应当有 $50\%$ 概率是 $1$。因为有 $10^4$ 个随机数，所以随机结果符合统计分布规律(即频率趋于概率)。那么任取一位，判断其 $1$ 出现的概率即可。比如可以取最末尾。

本题有非常多的解法。欢迎在评论区分享你的解法。

参考代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
ll p;
signed main()
{
    for (ll i = 0, x; i < 10000; ++i)
    {
        sc(x);
        p += x & 1;
    }
    printf("%c", abs(p - 7500) < abs(p - 5000) ? 'A' : 'B');
    return 0;
}
```

# 函数2

本蒟蒻想不出求精确解的方法。仅介绍求近似解的随机算法——模拟退火。

> 如果您有更好的解法或更好的实现欢迎评论区分享~

注意本题峰值太多，所以爬山算法不可用（爬山算法样例都过不了）

```cpp
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
typedef double db;
ll a, b, c, d, e, l, r;
db x, t, now, mx, ans = -6;
db f(db x)
{
    return sin(x / a) + sin(x / b) + sin(x / c) + sin(x / d) + sin(x / e);
}
void solve()
{
    static ll rd = 1;//也可以用random_device生成种子
    mt19937 mt(rd + time(0));
    rd *= 2;
    uniform_real_distribution<db> dist0(0, 1);
    now = (l + r) / 2, t = r - l;
    while (t > 1e-6)
    {
        uniform_real_distribution<db> dist(-t, t);
        db newx = now + dist(mt);
        newx = max(1. * l, min(1. * r, newx));
        db fnow = f(now), fnew = f(newx);
        db dt = (-fnew) - (-fnow);
        if (fnow < fnew || exp(-dt / t) > dist0(mt))
        {
            now = newx;
        }
        if (ans < fnew)
        {
            ans = fnew;
        }
        t *= 0.999;
    }
    for (ll i = 0; i < 1000; ++i) //在终温随机多次
    {
        uniform_real_distribution<db> dist(-t, t);
        db newx = now + dist(mt);
        newx = max(1. * l, min(1. * r, newx));
        db fnow = f(now), fnew = f(newx);
        if (fnow < fnew)
        {
            now = newx;
        }
        if (ans < fnew)
        {
            ans = fnew;
        }
    }
}
signed main()
{
```

```
    sc(a), sc(b), sc(c), sc(d), sc(e), sc(l), sc(r);
    for (ll i = 0; i < 20; ++i) //重要：多次退火
    {
        solve();
    }
    printf("%lf", ans);
    return 0;
}
```

# 珂朵莉树

介绍两种解法。(分块理论上也能做，感兴趣自行实现)

## 珂朵莉树解法

纯模板，不解释。仅适用于随机数据，非随机数据或初始不是常数列会 TLE 。复杂度 $O(n \log \log n)$

```
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
ll n;
struct node_t
{
    ll l, r;
    mutable ll v; // mutable使得const可变(set元素是const)
    node_t(const ll &il, const ll &ir, const ll &iv) : l(il), r(ir), v(iv)
{}
    inline bool operator<(const node_t &o) const { return l < o.l; }
};
set<node_t> odt; //建立一棵珂朵莉树
auto split(ll x) //珂朵莉基操
{
    if (x > n)
        return odt.end();
    auto it = --odt.upper_bound(node_t{x, 0, 0});
    if (it->l == x)
        return it;
    ll l = it->l, r = it->r, v = it->v;
    odt.erase(it);
    odt.insert(node_t(l, x - 1, v));
    return odt.insert(node_t(x, r, v)).first;
}
void assign(ll l, ll r, ll v) //区间赋值为v, 均摊O(loglogn)
{
    auto itr = split(r + 1), itl = split(l);
    odt.erase(itl, itr);
    odt.insert(node_t(l, r, v));
}
void add(ll l, ll r, ll v) //区间增加v, 均摊O(loglogn)
{
    auto itr = split(r + 1), itl = split(l);
    for (; itl != itr; ++itl) //枚举每个子区间
    {
        itl->v = itl->v + v;
```

```
    }
}
ll query(ll l, ll r) //区间查询, 均摊O(loglogn)
{
    ll res = 0;
    auto itr = split(r + 1), itl = split(l);
    for (; itl != itr; ++itl)
    { // itl->l, itl->r, itl->v 是当前子区间的左右端点和值
        res += (itl->r - itl->l + 1) * (itl->v);
    }
    return res;
}
signed main()
{
    sc(n);
    odt.insert({1, n, 1236895}); //初始化区间
    ll m, cmd, l, r, x;
    for (sc(m); m--;)
    {
        sc(cmd), sc(l), sc(r);
        if (cmd == 1)
        {
            sc(x), assign(l, r, x);
        }
        else if (cmd == 2)
        {
            sc(x), add(l, r, x);
        }
        else
        {
            printf("%lld\n", query(l, r));
        }
    }
    return 0;
}
```

## 线段树解法

可以认为也是线段树比较经典的模板题。适用面远大于珂朵莉树。

有两个不同的区间修改操作，它们会相互影响，所以要开两个懒标记

用 `laz1` 代表操作 1 的懒标记(初始为 0 ，0 代表没有需要执行的懒操作)，用 `laz2` 代表操作 2 的懒标记(初始也为 0)

对每次操作 1 ，区间赋值，把 `laz1` 设为 $x$ ，把 `laz2` 设为 $0$ ，因为赋值后加法全无效了

对每次操作 2 ，区间加法，把 `laz2` 加上 $x$

根据这个想法，需要对每次 pushdown 操作，先判 `laz1` ，如果发现 `laz1` 不为 0 ，就把 `laz1` 下传给两儿子，并且**把两儿子的 `laz2` 清零** (注意：这很重要)。判完之后，再判并下传 `laz2` 。

> 如果没有 `laz1` 下传的时候把两儿子 `laz2` 清零，在区间赋值时，两儿子可能会多执行赋值前遗留的区间加法，从而造成结果偏大

复杂度 $O((n+m)\log n)$

```cpp
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
#define mn 100010
#define lfs p << 1
#define rfs p << 1 | 1
#define mkcf ll cf = (lf + rf) >> 1
// laz1: lazy-assign ; laz2: lazy-add
ll t[mn << 2], laz1[mn << 2], laz2[mn << 2], a[mn];
void build(ll p, ll lf, ll rf)
{
    if (lf == rf)
    {
        t[p] = 1236895;
        return;
    }
    mkcf;
    build(lfs, lf, cf);
    build(rfs, cf + 1, rf);
    t[p] = t[lfs] + t[rfs];
}
void pushdown(ll p, ll lf, ll rf)
{
    mkcf;
    if (laz1[p])
    {
        t[lfs] = laz1[p] * (cf - lf + 1);
        t[rfs] = laz1[p] * (rf - cf);
        laz1[lfs] = laz1[rfs] = laz1[p];
        laz2[lfs] = laz2[rfs] = 0; //注意要清理laz2
        laz1[p] = 0;
    }

    t[lfs] += laz2[p] * (cf - lf + 1);
    t[rfs] += laz2[p] * (rf - cf);
    laz2[lfs] += laz2[p], laz2[rfs] += laz2[p];
    laz2[p] = 0;
}
void assign(ll p, ll lf, ll rf, ll lc, ll rc, ll x)
{
    if (lf >= lc && rf <= rc)
    {
        t[p] = (rf - lf + 1) * x;
        laz1[p] = x;
        laz2[p] = 0;
        return;
    }
    pushdown(p, lf, rf);
    mkcf;
    if (cf >= lc)
    {
        assign(lfs, lf, cf, lc, rc, x);
    }
    if (cf + 1 <= rc)
    {
        assign(rfs, cf + 1, rf, lc, rc, x);
    }
```

```
    t[p] = t[lfs] + t[rfs];
}
void add(ll p, ll lf, ll rf, ll lc, ll rc, ll x)
{
    if (lf >= lc && rf <= rc)
    {
        t[p] += (rf - lf + 1) * x;
        laz2[p] += x;
        return;
    }
    pushdown(p, lf, rf);
    mkcf;
    if (cf >= lc)
    {
        add(lfs, lf, cf, lc, rc, x);
    }
    if (cf + 1 <= rc)
    {
        add(rfs, cf + 1, rf, lc, rc, x);
    }
    t[p] = t[lfs] + t[rfs];
}
ll query(ll p, ll lf, ll rf, ll lc, ll rc)
{
    if (lf >= lc && rf <= rc)
    {
        return t[p];
    }
    pushdown(p, lf, rf);
    mkcf;
    ll res = 0;
    if (cf >= lc)
    {
        res += query(lfs, lf, cf, lc, rc);
    }
    if (cf + 1 <= rc)
    {
        res += query(rfs, cf + 1, rf, lc, rc);
    }
    t[p] = t[lfs] + t[rfs];
    return res;
}
signed main()
{
    ll n, m, l, r, x, cmd;
    sc(n), sc(m);
    build(1, 1, n);
    while (m--)
    {
        sc(cmd), sc(l), sc(r);
        if (cmd == 1)
        {
            sc(x);
            assign(1, 1, n, l, r, x);
        }
        else if (cmd == 2)
        {
            sc(x);
```

```
            add(1, 1, n, l, r, x);
        }
        else if (cmd == 3)
        {
            printf("%lld\n", query(1, 1, n, l, r));
        }
    }
    return 0;
}
```

# 骗红包

设 $f(n)$ 是有 $n$ 枚硬币时先手(zf)能赢得多少硬币，若先手必胜 $f(n)=n$，否则 $f(n)=0$

$n$ 服从 $[1,10^3]$ 内的均匀分布，所以期望为 $E=\frac{1}{1000}\sum_{i=1}^{n}f(i)$，即 $f(i)$ 均值

那么进行 $1000$ 轮游戏，$1000E=\sum_{i=1}^{n}f(i)$，所以求出 $f(i)$ 即可

不可以直接 DFS 搜索(有经验的话应该一眼TLE)，所以要记忆化搜索。

> TLE 的证明：
>
> 事实上对单次 $n$， 设 DFS 的执行次数为 $T(n)$，则：
>
> $$\begin{aligned}T(n)&=T(n-1)+T(\frac{n}{2})\\&=\sum_{i=1}^{n}T(\frac{i}{2})\\&=\sum_{i=1}^{n}\sum_{j=1}^{i}T(\frac{j}{4})\\&=\cdots\\&=O(n)\times O(\frac{n}{2})\times O(\frac{n}{4})\times\cdots\times O(\frac{n}{n})\\&=O(n^{\log_2 n})\end{aligned}$$
>
> 其中 $\log_2 n\approx 10$，是指数复杂度，所以不能爆搜。

记忆化搜索，我们发现每次只有两个状态，所以可以非递归搜索即可：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll sum, state[1024], N = 1, P = 0;
signed main()
{
    for (ll i = 1; i <= 1000; ++i)
    {
        if (state[i - 1] == P || state[i / 2] == P)
        {
            state[i] = N;
            sum += i;
        }
    }
    printf("%lld", sum);
    return 0;
```

```
}
```

## 灭鼠先锋 II

对给定局面，求后手是否必胜。由于 $2^{20} \approx 10^6$ ，遍历全部状态是可行的，且状态转移数不多(每个状态的转移数最多为 $nm + n(m-1)$ )，所以是一个稀疏图，可以考虑记忆化搜索，可以用二进制状态压缩存储每个局面的搜索结果，时间复杂度为 $O(nm2^{n+m})$ ，空间复杂度为 $O(2^{n+m})$ 。根据博弈论局面的性质直接DFS即可。

```
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
#define mn 1048580 // 2^{20}+4
#define N 0
#define P 1
ll n, m, t, s[mn]; // s[si]=N(0)先手必胜;s[si]=P(1)先手必败
#define O 0
#define X 1
struct state
{
    bool a[22][22];
    ll toi() //把a转化为二进制状态
    {
        ll r = 0;
        for (ll i = 1, k = 1; i <= n; ++i)
        {
            for (ll j = 1; j <= m; ++j, k *= 2)
            {
                r += k * a[i][j];
            }
        }
        return r;
    }
} now;
ll dfs()
{
    ll si = now.toi();
    if (s[si] != -1) //记忆化剪枝
    {
        return s[si];
    }
    ll hasP = 0, numx = 0; //是否可以移动到P
    for (ll i = 1; i <= n; ++i)
    {
        for (ll j = 1; j <= m; ++j)
        {
            numx += now.a[i][j] == X;
            if (now.a[i][j] == O)
            {
                now.a[i][j] = X;
                hasP |= dfs();
                now.a[i][j] = O; //回溯
            }
            if (j < m && now.a[i][j] == O && now.a[i][j + 1] == O)
```

```
            {
                now.a[i][j] = now.a[i][j + 1] = X;
                hasP |= dfs();
                now.a[i][j] = now.a[i][j + 1] = O;
            }
        }
    }
    s[si] = hasP ? N : (numx == n * m ? N : P);
    return s[si];
    //一般的ICG就hasP?N:P就行了
}
char ans[] = "LV", q[22][22]; //先手N(0)=V,P(1)=L,先手VL;故后手LV
signed main()
{
    for (ll i = 0; i < mn; ++i)
    {
        s[i] = -1; //表示未知
    }
    sc(n), sc(m);
    dfs();
    for (sc(t); t--;)
    {
        for (ll i = 1; i <= n; ++i)
        {
            scanf("%s", q[i] + 1);
        }
        for (ll i = 1; i <= n; ++i)
        {
            for (ll j = 1; j <= m; ++j)
            {
                now.a[i][j] = q[i][j] == 'O' ? O : X;
            }
        }
        printf("%c\n", ans[s[now.toi()]]);
    }
    return 0;
}
```

# 巴什博奕

当且仅当 $m+1|n$ (即 $n$ 是 $m+1$ 的倍数)先手必败，否则先手必胜。先手必胜策略为：每次先手行动都取若干石子使得剩下石子是 $m+1$ 的倍数，此时轮到后手的话后手无论如何都无法取得一种方案使得剩下石子是 $m+1$ 的倍数，再次轮到先手时先手重复上述步骤即可(显然最后先手取完后，剩余石子数 $0$ ，可以认为 $0$ 也是 $m+1$ 的倍数)。

~~什么？这题还需要参考代码？~~

# 锦乐游戏

由于 $2^x$ 都是偶数，所以 $2^x-1$ 都是奇数。因为每次取球都减少奇数个球，而奇数=奇数个奇数相加，偶数=偶数个奇数相加。奇数回合先手，偶数回合后手，所以 $n$ 为奇数先手胜， $n$ 为偶数后手胜。

~~什么？这题还需要参考代码？~~

# 取石子游戏

在 $n = 1, 2$ 下列出大量例子找规律，找出初步规律然后在 $n = 3, 4$ 下验证规律，最后发现只有石子堆个数为偶数且都是 $1$ 时 `sakiko` ，否则都是 `momoko`

即 `sum == n && n % 2 == 0` 。~~什么？这题还需要参考代码？~~

具体证明如下：

1. [1]是`momoko`必胜，[1,1]是`sakiko`必胜
2. 全为1时双方都只有一种取法，容易发现奇数个[1]最后一定是`momoko`取，`momoko`胜，同理偶数个是`sakiko`必胜
3. `[x]`(x>1)时，
    - 若x=2,`momoko`取2必胜；
    - 若x=3,`momoko`取1，`sakiko`只能取1，回到[1]，[1]是`momoko`必胜
    - 若x=y(y是偶数,y>2)，`momoko`取1,`sakiko`只能取1，回到x=y-2，不断如此操作，一定会到达x=2，此时`momoko`取2必胜；
    - 若x=y(y是奇数,y>3)，`momoko`取1,`sakiko`只能取1，回到x=y-2，不断如此操作，一定会到达x=3，此时`momoko`取1,`sakiko`取1，`momoko`取1必胜。
    - 综上所述，[x]时，`momoko`必胜
4. [a1, a2, a3, ...]时
    - 第一回合`momoko`对任意一个数按照(3)的方法取一次；
    - 在这之后，`sakiko`无论取哪个数，`momoko`都可以按照(3)的方法取一次；
    - 如果每个数都大于1，那么一定可以按照(3)的方法让`momoko`对每个数都是它最后取完的
    - 如果大于1个数字的数目是1，`sakiko`一定可以取完至少一次1。但是对不是1的数，按照(3)的方法，`momoko`至少存在一种策略，使得`sakiko`没有办法取完，所以只要在这a1,a2,a3...里存在不是1的数，`momoko`一定可以最后取完它

综上所述，当且仅当有偶数个1时`sakiko`必胜，否则`momoko`必胜。

# 异世界的人偶训练

请参考2021蓝桥杯热身赛#2比赛题解。

# 云烟蓝星对决

请参考2022天梯赛选拔赛比赛题解。

# 高僧斗法

把和尚两两配对，如 $1, 2$ 配对； $3, 4$ 配对。每对和尚之间相隔多少个台阶就是多少个石子，配对的一对和尚就是一个石子堆。那么我们就把原题目转化为了 Nim 游戏。

如果是共有奇数个和尚，那么落单的那个可以认为是数量为 $0$ 的石子堆。

每次取石子 $x$ 个，等效于将配对的第一个和尚(即第奇数个)向前移动 $x$ 步。

两两配对的正确性证明：

如果移动随便一个和尚 $p$ 步，那么下一个人可以跟随着移动它的上一个和尚 $p$ 步，不断这么操作。这样的情况递归下去就是，假设移动第 $i$ 个和尚 $p$ 步，那么接下来 $i$ 步会使得前 $i$ 个和尚都往前移动了 $p$ 步。当 $i$ 是偶数时，可以保证在局面(先后手次序)不变的情况下，使得整体往右平移，最终使得第偶数个和尚与下一个和尚的距离都等价变成 $0$ 。即所有第偶数个和尚都无法移动为止。此时只有第奇数个和尚是能够移动的。

对共有奇数个和尚时，因为最后一个和尚的位置按照题意本来就是最高级台阶，他本来就是不能移动的，不用管他即可。

因此，转化后，根据 Nim 游戏规律，异或和为 0 先手必败，否则先手必胜。

因为数据量很小，我们可以暴力枚举所有第一步移动状态，枚举所有的将第 $i$ 个和尚移动 $j$ 步，重新计算间隔，然后看看这么做之后新的 Nim 游戏石子堆异或和是不是为 0 ，是的话走完先手必败，即没走先手必胜。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define mn 102
int m[mn],n,v,s[mn],sum;
signed main()
{
    while(scanf("%d",&v)!=EOF) m[++n]=v;
    for(int i=1;i<n;++i) s[i]=m[i+1]-m[i]-1;
    for(int i=1;i<n;i+=2) sum^=s[i];
    if(sum==0) return printf("-1")&0;
    for(int i=1;i<n;++i) for(int j=1;m[i]+j<m[i+1];++j)
    {
        s[i]-=j;//将第i个和尚走j步
        if(i!=1) s[i-1]+=j;//对应影响
        sum=0;
        for(int k=1;k<n;k+=2) sum^=s[k];
        if(sum==0) return printf("%d %d",m[i],m[i]+j)&0;
        s[i]+=j;//回溯
        if(i!=-1) s[i-1]-=j;
    }
    return 0;//程序代码不可能走到这里
}
```