枚举优化与搜索 习题讲解

洛谷网校

基础衔接提高计划

洛谷

www.luogu.com.cn

乘积

对于一个给定自然数 n，求出将其表示为若干个不为 1 的斐波那契数的乘积的方案数

$2 \leq n \leq 10^{18}$

- 虽然看起来数据范围很大，但我们仔细观察就可以发现，斐波那契数的增长速度非常快，几乎是以指数趋势。简单计算可以得知 10^{18} 范围内只有不超过 86 个斐波那契数。
- 这样我们就可以开始搜索了，不妨记第 i 个斐波那契数为 $fib[i]$，我们用 $dfs(n,i)$ 计算用小于等于 $fib[i]$ 的斐波那契数拆分 n 的方案数。则考虑对于 $fib[i]$ 这个数，如果其不能整除 n，则 $dfs(n,i)=dfs(n,i-1)$。否则，n 可以被 $fib[i]$ 整除，$dfs(n,i)=dfs\left(\frac{n}{fib[i]},i\right)+dfs(n,i-1)$
- 初始时计算 $dfs(n,86)$

```
ll fib[90]; ll T, n;
ll dfs(ll n, ll i) {
    if (n == 1) return 1;
        //n等于1时拆分完成
    if (i <= 1) return 0;
        //我们只用大于1的斐波那契数进行拆分
    if (!(n % fib[i]))
        return dfs(n/fib[i], i)+dfs(n, i-1);
    else
        return dfs(n, i-1);
}
```

```
int main() {
    scanf("%lld", &T);
    fib[0]=fib[1]=1;
    for (ll i = 2;i <= 86; i++)
        fib[i] = fib[i-1] + fib[i-2];
        //预处理斐波那契数
    while(T--) {
        scanf("%lld", &n);
        printf("%lld\n", dfs(n,86));
    }
    return 0;
}
```

奇怪的按钮 2.0

有两个正整数 a 和 b，而你有一个奇怪的按钮，每次可以选择以下三种操作之一：

- $a \leftarrow a \times 2$
- $b \leftarrow b - 1$
- $b \leftarrow b + 1$

求使得 $a = b$ 的最小操作次数。

$1 \leq a, b \leq 10^9$

- 考虑最朴素的做法，我们暴力枚举每个操作需要进行多少次，然后模拟在进行这些操作之后，a 和 b 是否相等
- 进一步的，我们发现其实在操作数最小时，二、三操作只会存在一个。因为一次 $b+1$ 会和一次 $b-1$ 相抵消。更具体的，在不使用一操作的前提下，使得 a 和 b 相等的最小操作数就是 $|a-b|$ 。
- 于是，我们直接枚举一操作的次数 k ，此时 $a \leftarrow a \times 2^k$ ，则还需 $\left|a \times 2^k - b\right|$ 次二、三操作可使 a 和 b 相等。取所有 k 中操作数最小的即可。

洛谷

```cpp
#include<bits/stdc++.h>
using namespace std;
#define int long long
int a,b,ans=1e9;
signed main()
{
    cin>>a>>b;
    for(int k=0;k<=31;++k)
    {
        int tmp = 1<<k; // 2的k次方
        ans = min(ans, k+ abs(a*tmp - b));
              // 计算操作次数
    }
    cout<<ans<<endl;
       // 输出最小操作次数
    return 0;
}
```

淹园2.0

给定一个 $N \times N$ 的矩阵，其中 `.` 表示积水、 `#` 表示干地。

其中通过 “上下左右” 四个方向上连在一起的一片干地组成一块 “安全地带” 。并且如果一块干地与积水相邻，其在一天后会变为积水。

请你计算：一天后会有多少安全地带完全变为积水。

$1 \leq \mathrm{N} \leq 1000$

- 首先我们考虑什么时候一个安全地带不会完全变为积水，注意到如果一个安全地带中，存在一个干地，其上下左右**在此时**都不是积水的话，那一天之后其一定不会变为积水，也就意味着这个安全地带不会完全变为积水
- 于是我们使用dfs遍历安全地带（连通块）的每个干地，如果存在一天后不会变为积水的干地，那么这个安全地带不会完全变为积水。相反，如果不存在，则其完全变为积水，我们将答案加一。
- 注意不能先将一天后的地图绘制后统计剩下多少个连通块，因为这有可能会使得原先的一个连通块分成多个连通块，从而统计错误。

```
int n,pre,suc,vis[1010][1010];
int dx[]={0,1,0,-1},dy[]={1,0,-1,0};
char mp[1010][1010];
int dfs(int x,int y) {
    vis[x][y]=1;//标记
    int cnt=0,tag=0;
    for(int i=0;i<4;i++)
        if(mp[x+dx[i]][y+dy[i]]!='.')
            cnt++;
    if(cnt==4)//该干地四周无积水
        tag = 1;

    for(int i=0;i<4;i++){
        int xx=x+dx[i],yy=y+dy[i];
        if(mp[xx][yy]!='#'||vis[xx][yy])
            continue;
        if (dfs(xx,yy))
            tag = 1; //存在一个干地周围无积水
    }
    return tag;
}
```

```
int main(){
    cin>>n;
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            cin>>mp[i][j];
    for(int i=1;i<n-1;i++)
        for(int j=1;j<n-1;j++)
            if(mp[i][j]=='#'&&!vis[i][j]){
                //如果是干地且未被访问
                pre++;
                suc+=dfs(i,j);
            }
    cout<<pre-suc;//总数量-没变为积水的
}
```

辐射

在一条数轴表示的直线道路上，有 N 个污染源。每个污染源的位置按从左到右依次为 $A_1 < A_2 < \cdots < A_N$。

我们定义某个位置 x 的“污染级别”为该位置到所有污染源之间距离的最小值。

给定一个整数 L，我们关心从 $x = 0$ 到 $x = L$ 这 $L + 1$ 个整数位置的污染级别。请你编程，输出其中按污染级别从小到大排序后的前 K 小的值。

$1 \le L \le 10^{18}, 1 \le N \le 3 \times 10^5, 1 \le K \le \min(5 \times 10^5, L + 1)$

- 最为简单的想法是遍历 $[0, L]$ 的每一个位置，计算其到各个污染源的距离然后输出其中前 K 小的。但是 L 过大，无法通过
- 但是我们注意到 K 和 N 很小。由于本质是要计算到这 N 个污染源距离的最小的 K 个点，可以直接通过 bfs 计算（bfs 保证第一次到达时一定是距离最短的，且一定是由距离从小到大的顺序进行遍历）。我们一开始将所有污染源加入队列，然后每次将队头的点弹出，输出其距离，并将旁边的没入队的点加入队列。是否入队过的信息通过map维护即可。

```cpp
const int N = 3e5 + 10;
int n,k;
ll L,a[N];
unordered_map <ll,bool> vis; // 入队标记
queue<ll> q,dis; // bfs队列
int main(){
    cin >> L >> n >> k;
    for(int i = 1;i <= n;i++)
        cin >> a[i];
    for(int i = 1;i <= n;i++){
        q.push(a[i]); // 将所有污染入队列
        dis.push(0); // 初始距离为0
        vis[a[i]] = true; // 标记已入队
    }
    while(k--){
        ll x = q.front(); q.pop();
        // 取出队首位置
        ll d = dis.front(); dis.pop();
        printf("%lld\n", d);
        // 取出队首距离并输出

        if (x + 1 <= L && !vis[x + 1]) {
            vis[x + 1] = true;
            q.push(x + 1); // 入队
            dis.push(d + 1); // 距离加一
        }
        if (x - 1 >= 0 && !vis[x - 1]) {
            vis[x - 1] = true;
            q.push(x - 1); // 入队
            dis.push(d + 1); // 距离加一
        }
    }
    return 0;
}
```

P8318 『JROI-4』淘气的猴子

有一个长度为 n 的序列 a，对这个序列进行 m 次操作，得到序列 b，操作分为以下两种类型：

- $1\ x\ y$ 表示第 x 个元素加上第 y 个元素
- $2\ x\ y$ 表示第 x 个元素乘上第 y 个元素

现给出操作后的序列 b 以及 m 次操作，问一开始的序列 a

- 正向考虑似乎很难进行，我们不妨考虑反向枚举操作，一步一步恢复原本的数组
- 具体来说，我们先用一个结构体数组将每一步的操作记录下来，然后从后往前枚举操作，并进行恢复。同时注意 $x = y$ 的情况即可
  - 对于 $1\ x\ y$，若 $x \neq y$ 我们使得 $b_x = b_x - b_y$，否则使得 $b_x = \frac{b_x}{2}$
  - 对于 $2\ x\ y$，若 $x \neq y$ 我们使得 $b_x = \frac{b_x}{b_y}$，否则使得 $b_x = \sqrt{b\_x}$

```
struct data{
    int t,x,y;
    //记录操作
}f[N];

int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        cin>>b[i];
    for(int i=1;i<=m;i++)//记录操作
        cin>>f[i].t>>f[i].x>>f[i].y;
```

```
    for(int i=m;i>0;i--){
        //从后往前进行恢复
        ll t=f[i].t,x=f[i].x,y=f[i].y;
        if(t==1){
            //如果是加法
            if(x!=y)b[x]-=b[y];
            else b[x]/=2;
        }else if(t==2){
            //如果是乘法
            if(x!=y)b[x]/=b[y];
            else b[x]=sqrt(b[x]);
        }
    }
    for(int i=1;i<=n;i++)
        printf("%lld ",b[i]);
    return 0;
}
```

P8185 [USACO22FEB] Blocks B

有一套四块木块，其中每块都是一个立方体，六面各写着一个字母。可以通过将木块排成一排使得木块顶部的字母拼出单词。

给定四个木块上的字母，以及要拼写的单词列表，请确定列表中哪些单词可使用木块成功拼写

- 注意到木块的个数实际上很少，考虑直接枚举选择哪些木块以及木块的摆放顺序
- 确定木块的摆放顺序之后，我们已经可以将每个木块与其应当朝上的字符一一对应，并判定该木块是否有对应的字符了
- 不过由于题目数据范围较少，直接枚举每个木块朝上的一面是哪个字符，最后判断拼出的单词和期望的单词是否一致也完全能通过。

洛谷

```
char cub[4][10],s[10],tmp[10];
int vis[4],len;
int dfs(int step) {
    if (step == len) {
        // 检验拼出的字符串是否与s相同
        for (int i = 0; i < len; i++)
            if (tmp[i] != s[i])
                return 0; //不匹配
        return 1;
    }
    for (int i = 0; i < 4; i++) {
        if (!vis[i]) {
            // 确定选择的立方体
            vis[i] = 1;
            for(int j=0;j<6;j++){
                // 尝试每个面的字符
                tmp[step] = cub[i][j];
                if (dfs(step + 1))
                    return 1;
            }
            vis[i] = 0;
        }
    }
    return 0;
}
int main(){
    int n; cin >> n;
    for(int i=0;i<4;i++)
        cin >> cub[i];
    while(n--){
        cin>>s;
        len = strlen(s);
        memset(vis,0,sizeof(vis));
        if (dfs(0))
            puts("YES");
        else
            puts("NO");
    }
    return 0;
}
```

P6068 『MdOI R1』GCD? GCD!

给定一个正整数 n，把它分成三个 **互不相等的** 正整数 a, b, c 之和，使得 $\gcd(a, b, c)$ 最大。

对于 100% 的数据，$1 \leq \mathrm{T} \leq 100, 1 \leq n \leq 10^9$。其中 T 表示数据组数。

- 首先考虑最朴素的做法，我们直接暴力枚举 a,b,c 再计算 $\gcd(a,b,c)$，选取其中最大的。但这样做显然无法通过全部数据。
- 接着我们考虑直接枚举 $t=\gcd(a,b,c)$，当然我们应该要求 n 被 t 整除。此时会发现，存在一组不同的 a,b,c 使其加起来等于 n，当且仅当 $\frac{n}{t}\geq 6$。
  - 若 $\frac{n}{t}\geq 6$，则直接取 $a=t,b=2t,c=\left(\frac{n}{t}-3\right)\geq 3t$，合法
  - 若存在 $a=xt,b=yt,c=zt$，不妨设 $a\leq b\leq c$，则 $x\geq 1,y\geq 2,z\geq 3$，也即 $\frac{n}{t}=x+y+z\geq 6$
- 于是我们从 1 到 $\frac{n}{6}$ 枚举 t，取其中最大的合法的。

- 但是由于题目中 $n \leq 10^9$，这样做仍然不足以通过。我们观察发现实际上这个枚举过程中有很多重复的部分。对于我们枚举的一个 n 的因数 t，其对应的 $\frac{n}{t}$ 肯定也是 n 的因数。这样我们直接不枚举大于 $\sqrt{n}$ 的 t，因为其必然等于某些 $\frac{n}{t'}$ 其中 $t' \leq \sqrt{n}$。在计算 t'的同时计算 $\frac{n}{t'}$ 的贡献即可。整体复杂度 $O(T\sqrt{n})$

洛谷

```cpp
int main(){
    ll T,n; cin>>T;
    while(T--){
        cin>>n;
        ll ans = -1;
        //注意无解的情况
        for(ll t=1;t*t<=n;t++){
            if (n%t)
                continue;
            //取t为gcd
            if (n/t>=6)
                ans = max(ans,t);
            //取n/t为gcd
            if (t>=6)
                ans = max(ans,n/t);
        }
        cout<<ans<<endl;
    }
    return 0;
}
```

P1003 [NOIP2011 提高组] 铺地毯

为了准备一个独特的颁奖典礼，组织者在会场的一片矩形区域（可看做是平面直角坐标系的第一象限）铺上一些矩形地毯。一共有 n 张地毯，编号从 1 到 n 。现在将这些地毯按照编号从小到大的顺序平行于坐标轴先后铺设，后铺的地毯覆盖在前面已经铺好的地毯之上。

地毯铺设完成后，组织者想知道覆盖地面某个点的最上面的那张地毯的编号。注意：在矩形地毯边界和四个顶点上的点也算被地毯覆盖。地毯的左下角的坐标设为 (a,b) 。

$1 \le n \le 10^4$，$0 \le a,b \le 10^5$

- 考虑较为暴力的做法，我们直接模拟铺地毯的过程。开一个二维数组记录地毯覆盖的状态。每新加入一个地毯，我们将其对应的矩形区域内全部赋值为他的id，这样该二维数组中每个位置都记录其上最后铺设的（即最上层的）地毯的id，查询时直接输出即可。但由于内存限制，这种做法无法通过。
- 注意到其实查询操作只有一次，我们不需要追求 $O(1)$ 查询。考虑按先后顺序枚举每个地毯，如果该地毯覆盖最终的查询位置，则更新最终答案。因为我们从前到后枚举，所以最终答案一定是最上层的地毯。时间复杂度 $O(n)$。

洛谷

```
int main(){
    int n=read();
    for(int i=1;i<=n;i++)
        a[i]=read(),b[i]=read(),g[i]=read(),k[i]=read();
    int x=read(),y=read();
    int ans=-1;
    for(int i=1;i<=n;i++)
        if ((x>=a[i])&&(x<=a[i]+g[i])
            &&(y>=b[i])&&(y<=b[i]+k[i])) ans=i;
    cout<<ans<<endl;
    return 0;
}
```

P2329 [SCOI2005] 栅栏

农夫约翰需要 n 块特定规格的木材，可是木材店老板说他这里只剩下 m 块大规格的木板了。不过约翰可以购买这些木板，然后切割成他所需要的规格。而且约翰有一把神奇的锯子，用它来锯木板，不会产生任何损失，也就是说长度为 10 的木板可以切成长度为 8 和 2 的两个木板。

你的任务：给你约翰所需要的木板的规格，还有木材店老板能够给出的木材的规格，求约翰最多能够得到多少他所需要的木板。

$m \leq 50, n \leq 1000$，木材规格小于 32767 。

- 首先我们考虑最朴素的做法。直接递归枚举每个约翰需要的木板切割自哪块大木材（或者未获得该木板），并从中统计切割出木板最多的方案。但这种做法的复杂度（约 $O(m^n)$）与目标数据范围相差甚远，只是简单的剪枝无法通过。
- 这时候，我们可以发现一个简单的结论：我们的最终方案一定可以只选择最短的那些木板，并且如果我们能获取 k 块木板，则一定可以获得 k' $(k' \leq k)$ 块木板。这样，我们可以通过二分答案 k，将问题转化为是否能够从这些木材中切割出最短的那 k 块木板。

- 这时候我们就可以考虑对搜索进行剪枝了。由于此处我们已将问题转为判定问题，故基本考虑可行性剪枝。
- 首先，如果我们剩下的木材的长度和已经小于需要的木板的长度和，则一定不可能切割成功，可以结束分支。
- 接着，我们可以考虑切割过程中“浪费”的部分。如果一段木材的长度小于最短的木板的长度，则其一定不可能被利用，我们可以将他的长度从木材长度和中扣除掉，从而得到“可利用的木材长度和”，帮助减少搜索时间。
- 最后，注意枚举顺序。从较长的木板开始枚举可以更早的引出矛盾从而剪枝，进而大幅优化效率。

```
int wast=0,sum=0;
bool check(int now,int need){
    if (now==0)return 1;
    if (need>sum-wast)return 0;
    for(int i=m;i>=1;i--){
        if (b[now]>a[i])continue;
        a[i]-=b[now];
        if(a[i]<b[1])wast+=a[i];
        int res=check(now-1,need);
        if (a[i]<b[1])wast-=a[i];
        a[i]+=b[now];
        if (res) return 1;
    }
    return 0;
}
int main(){
    m=read();
    for(int i=1;i<=m;i++)
        a[i]=read(),sum+=a[i];
    n=read();
    for(int i=1;i<=n;i++)
        b[i]=read();
    sort(a+1,a+m+1);sort(b+1,b+n+1);
    if(b[1]>a[m]){puts("0"); return 0;}
    for(int i=1;i<=n;i++)
        c[i]=c[i-1]+b[i];
    while(c[n]>sum)n--;
    int l=1,r=n,ans;
    while(l<=r){
        int mid=(l+r)/2;
        if (check(mid,c[mid]))
            ans=mid,l=mid+1;
        else r=mid-1;
    }
    cout<<ans<<endl;
    return 0;
}
```