W4 作业讲评

2025-02-22

览遍千秋

洛谷

www.luogu.com.cn

洛谷

目录

P7076 动物园

P7076 [CSP-S2020] 动物园

P7076 动物园

什么时候可以饲养编号为奇数的动物？

编号为奇数的动物的编号二进制的第 0 位为 1，所以如果能养编号为奇数的动物，m 条限制中有关第 0 位的饲料都要买。

换句话说，只有第 0 位相关的全部饲料都有才能养编号二进制第 0 位为 1 的动物。

推广这个结论，只有第 k 位相关的全部饲料都有才能养编号二进制第 k 位为 1 的动物。

P7076 动物园

所以根据饲料情况求出有多少个位可以为 1，假设有 p 位为 1，那么可以饲养的动物总数即为 2^p（每一位自由选择 $0/1$，其他位均为 0。）

具体的，求出现有的关系中每一位需要买的饲料，然后判断该位所需要买的饲料是否全都有，如果有则该位可以为 1，否则不能。

P7076 动物园

根据饲料情况求出有多少个位可以为 1，假设有 p 位可以为 1，那么可以饲养的动物总数即为 2^p（每一位自由选择 $0/1$，其他位均为 0。）

根据现有的动物编号的按位或结果，我们知道了现有的动物中哪些位为 1。

再根据饲料要求即可求出现有的饲料情况。

然后枚举每一位，判定这一位需要的所有饲料是否全部购买，如果全部购买则这一位可以为 1。

最终答案即为 $2^p - n$。

注意判断 $p = 64$ 的情况!

```
ull all=0,x;
scanf("%d%d%d%d",&n,&m,&c,&k);
for(int i=1;i<=n;i++)
{
    scanf("%llu",&x);
    all|=x;
}
int p,qq;
for(int i=1;i<=m;i++)
{
    scanf("%d%d",&p,&qq);
    q[p].push_back(qq);
}
for(int i=0;i<k;i++)
if((all>>i)&1)
{
    for(auto val:q[i])
        fd.push_back(val);
}
```

```
sort(fd.begin(),fd.end());
int cnt=0;
for(int i=0;i<k;i++)
{
    bool fl=false;
    for(auto val:q[i])
    {
        auto it=lower_bound(fd.begin(),fd.end(),val);
        if(it==fd.end()||(*it)!=val)
            {fl=true;break;}
    }
    if(!fl)
        ++cnt;
}
if(n==0&&cnt==64)
    printf("18446744073709551616\n");
else
{
    if(cnt==64)
        printf("%llu\n",(1ull<<63)-n+(1ull<<63));
    else
        printf("%llu\n",(1ull<<cnt)-n);
}
```

P9236 异或和之和

P9236 [蓝桥杯 2023 省 A] 异或和之和

P9236 异或和之和

我们可以利用前缀异或来表示区间异或

$$sum_i = a_1 \oplus a_2 \oplus a_3 \cdots a_i$$

区间 $[L, R]$ 的异或和 $a_L \oplus a_{L+1} \oplus a_{L+2} \cdots a_R = sum_R \oplus sum_{L-1}$

P9236 异或和之和

我们可以利用前缀异或来表示区间异或

$$sum_i = a_1 \oplus a_2 \oplus a_3 \cdots a_i$$

区间 $[L, R]$ 的异或和 $a_L \oplus a_{L+1} \oplus a_{L+2} \cdots a_R = sum_R \oplus sum_{L-1}$

所以有个暴力的做法是枚举所有的区间累加 $sum_R \oplus sum_{L-1}$，时间复杂度为 $O(n^2)$

P9236 异或和之和

区间 $[L,R]$ 的异或和 $a_L \oplus a_{L+1} \oplus a_{L+2} \cdots a_R = sum_R \oplus sum_{L-1}$

暴力的做法是枚举所有的区间累加 $sum_R \oplus sum_{L-1}$，时间复杂度为 $O(n^2)$

可以拆位考虑。

二进制的拆分原理与异或的计算

P9236 异或和之和

可以拆位考虑。

假设当前考虑的二进制位为 2^k 那一位，那么如果有 p 个区间满足 $sum_R \oplus sum_{L-1}$ 的第 k 位为 1，答案就会因为这一个二进制位增大 $2^k \times p$。

接下来考虑计算有多少个区间满足 $sum_R \oplus sum_{L-1}$ 的第 k 位为 1。

P9236 异或和之和

接下来考虑计算有多少个区间满足 $sum_R \oplus sum_{L-1}$的第 k 位为 1。

枚举 sum_R，如果 sum_R 第 k 位为 1，那么当 sum_{L-1} 的第 k 位为 0 的时候异或结果上第 k 位就为 1。反之亦然。

所以枚举 k，统计每个 sum_i 第 k 位上的 $0/1$ 个数即可计算。

洛谷

```
for(int k=0;k<=20;k++)
{
    z0=1;//sum[0]=0;
    z1=0;
    for(int i=1;i<=n;i++)
    {
        if((sum[i]>>k)&1)
        {
            ans+=(1ll<<k)*z0;
            z1++;
        }
        else
        {
            ans+=(1ll<<k)*z1;
            z0++;
        }
    }
}
printf("%lld\n",ans);
```

P5657 格雷码

P5657 [CSP-S2019] 格雷码

采用分治的思想，考虑 n 位格雷码的前 2^{n-1} 个第一位为 0，后 2^{n-1} 个第一位为 1。

所以如果 $k \leq 2^{n-1}$，就说明第 k 个格雷码第一位是 0，剩下的部分是 $n-1$ 位格雷码中的第 k 个。

如果 $k > 2^{n-1}$，说明第 k 个格雷码第一位是 1，剩下的部分是 $n-1$ 位格雷码按逆序排列的第 $k-2^{n-1}$ 个，也就是 $n-1$ 位格雷码的第 $2^{n-1}-(k-2^{n-1})+1=2^n-k+1$ 个。

P5657 格雷码

所以如果 $k < 2^{n-1}$，就说明第 k 个格雷码第一位是 0，剩下的部分是 $n-1$ 位格雷码中的第 k 个。

如果 $k \geq 2^{n-1}$，说明第 k 个格雷码第一位是 1，剩下的部分是 $n-1$ 位格雷码按逆序排列的第 $k-2^{n-1}$ 个，也就是 $n-1$ 位格雷码的第 $2^{n-1}-(k-2^{n-1})-1=2^n-k-1$ 个。

设计分治函数 $f(n,k)$ 表示要求 n 位格雷码的第 k 个。

进行递归求解。

当 $k \leq 2^{n-1}$ 时，答案等于 $0+f(n-1,k)$，这里的加号表示拼接。

当 $k > 2^{n-1}$ 时，答案等于 $1+f(n-1,2^n-k-1)$。

递归边界在 $n=1$ 的时候，根据 $k=0/1$ 得到一位格雷码。

```
string f(int n,unsigned long long k)
{
    if(n==1)
    {
        if(k==0) return "0";
        return "1";
    }
    if(k>=(1ull<<(n-1)))//k=(1ull<<(n-1))+tmp
    {
        return "1"+f(n-1,(1ull<<(n-1))-(k-(1ull<<(n-1)))-1);
    }
    else
        return "0"+f(n-1,k);
}
```

注意防止溢出!

P1885 Moo

P1885 Moo

P1885 Moo

P1885 Moo

注意：

可以发现第 k 个字符串是由

第 $k-1$ 个字符串 $+$ m $+$ $(k+2)$ 个 o $+$ 第 $k-1$ 个字符串

拼接而成。

可以通过递推求出第 i 个字符串长 $len[i]$

P1885 Moo

第 k 个字符串=

第 $k-1$ 个字符串 $+$ $m+(k+2)$ 个 $o+$ 第 $k-1$ 个字符串

考虑第 N 个字符落在哪个部分

P1885 Moo

第 k 个字符串=

第 $k-1$ 个字符串 + m + $(k+2)$ 个 o + 第 $k-1$ 个字符串

设第 x 个字符串长度为 len_x

考虑第 N 个字符落在哪个部分

$N \le len_{k-1}$，第一部分

$len_{k-1} < N \le len_{k-1} + k + 3$，第二部分

$len_{k-1} + k + 3 < N \le 2len_{k-1} + k + 3$，第三部分

P1885 Moo

第 k 个字符串=
第 $k-1$ 个字符串 + m + $(k+2)$ 个 o + 第 $k-1$ 个字符串

设第 x 个字符串长度为 len_x
考虑第 N 个字符落在哪个部分
$N \le len_{k-1}$，第一部分
相当于第 $k-1$ 个字符串的第 N 个字符。
$len_{k-1} < N \le len_{k-1} + k + 3$，第二部分
如果 $k = len_{k-1} + 1$ 那么就是 m，否则是 o。
$len_{k-1} + k + 3 < N \le 2len_{k-1} + k + 3$，第三部分
相当于第 $k-1$ 个字符串的第 $N - (len_{k-1} + k + 3)$ 个字符。

P1885 Moo

设第 x 个字符串长度为 len_x。

有 $len_x = 2len_{x-1} + x + 3$

考虑第 N 个字符落在哪个部分。

递归分治求出答案即可。

```
char f(int n,int x)
{
    if(n==0)
    {
        if(x==1) return 'm';
        else return 'o';
    }
    //len[n-1]+1+(n+2)+len[n-1]
    if(x>len[n-1]&&x<=len[n-1]+n+3)
    {
        if(x==len[n-1]+1) return 'm';
        else return 'o';
    }
    else if(x<=len[n-1]) return f(n-1,x);
    else return f(n-1,x-(len[n-1]+n+3));
}
```

P7167 Fountain

P7167 [eJOI2020 Day1] Fountain

P7167 Fountain

P7167 [eJOI2020 Day1] Fountain

先求出每个圆盘之后第一个大于自己的圆盘是哪一个

P7167 Fountain

先求出每个圆盘之后第一个大于自己的圆盘是哪一个

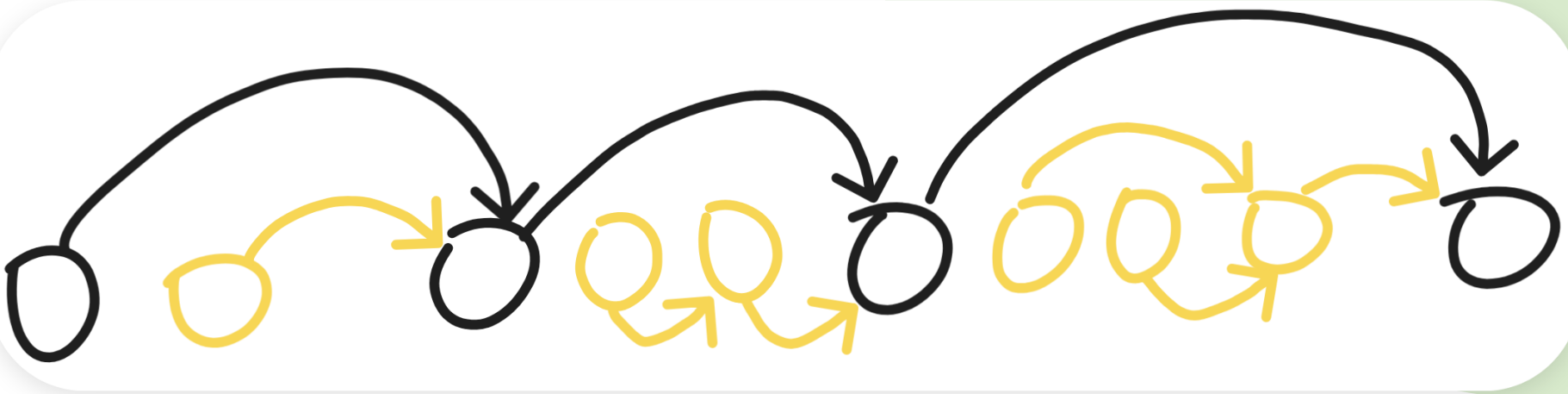
建立圆盘半径的最大值 ST 表，利用倍增求每个圆盘之后第一个大于自己的圆盘是哪一个。

建立圆盘半径的最大值 ST 表，利用倍增求每个圆盘之后第一个大于自己的圆盘是哪一个。

```
for(int i=1;i<=N;i++)
{
    int pos=i+1;//[i,pos-1] <=R[i]
    for(int k=J;k>=0;k--)
    {
        if(pos+(1<<k)-1<=N&&mx[k][pos]<=R[i])
        //[pos,pos+2^k-1] <=R[i]
            pos+=(1<<k);
    }
    //第一个大于自己的圆盘位置即为pos
    //没有圆盘大于自己的时候pos=N+1
}
```

P7167 Fountain

建立圆盘半径的最大值 ST 表，利用倍增求每个圆盘之后第一个大于自己的圆盘是哪一个。



让每一个圆盘指向自己后面第一个大于自己的圆盘。

或者说每个圆盘指向往后跳一步会跳到的圆盘。

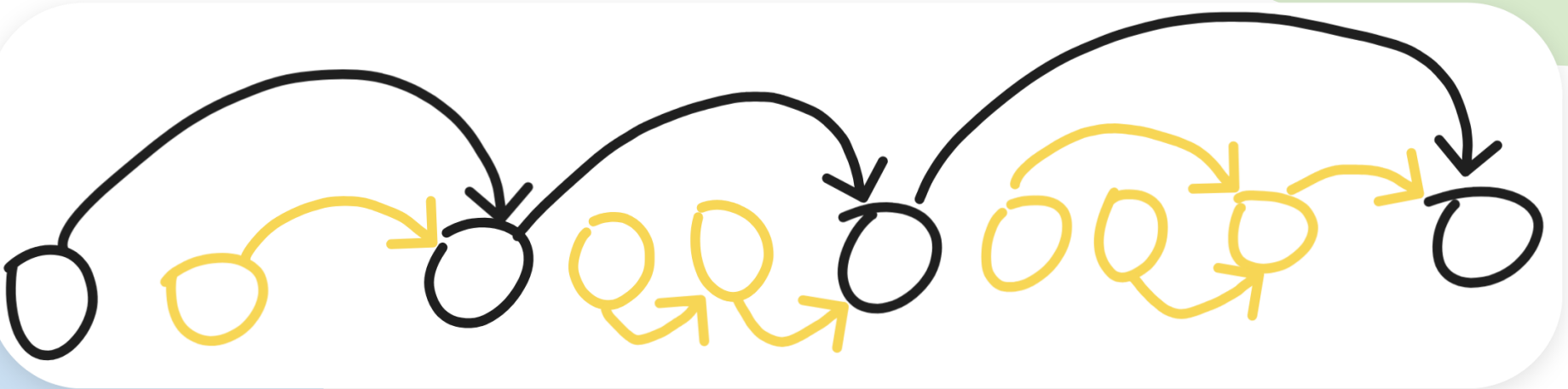
构成树关系

P7167 Fountain

让每一个圆盘指向自己后面第一个大于自己的圆盘。

或者说每个圆盘指向往后跳一步会跳到的圆盘。

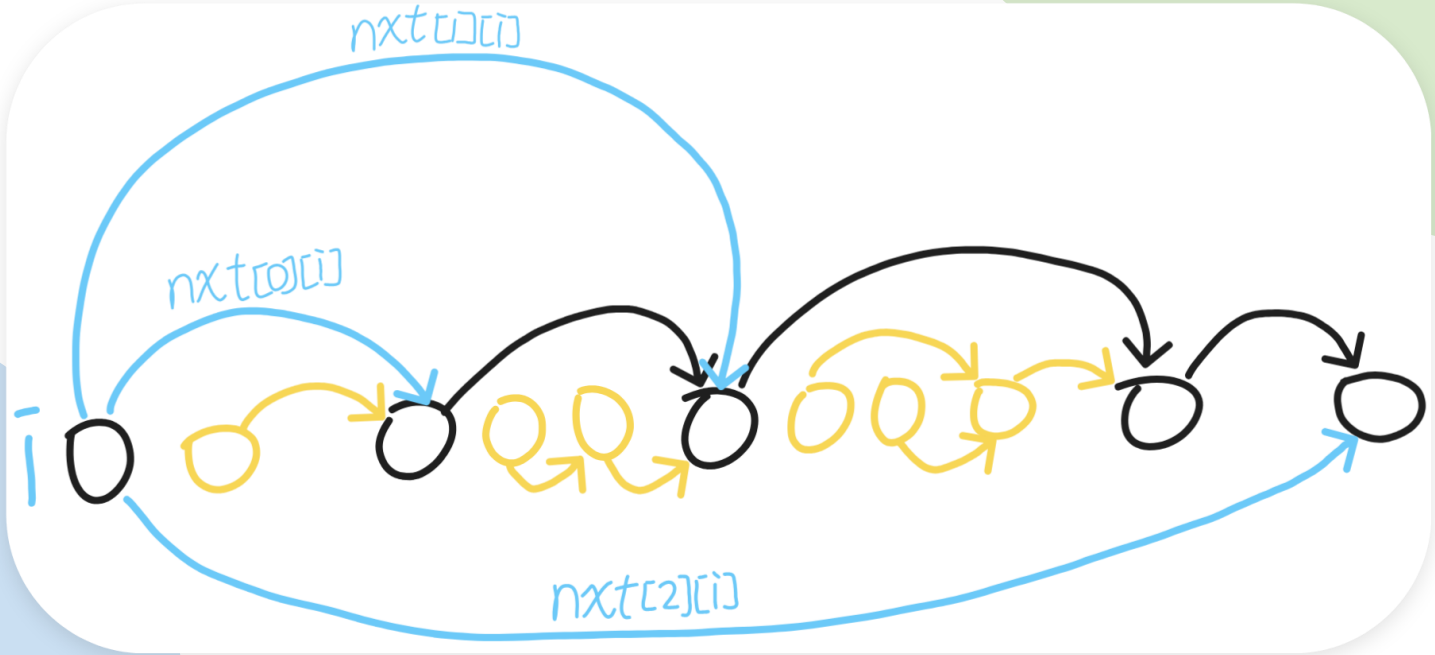
所求即，最少往后跳多少个圆盘能让跳过的圆盘容积之和不小于一开始水的体积。



P7167 Fountain

所求即，最少往后跳多少个圆盘能让跳过的圆盘容积之和不小于一开始水的体积。

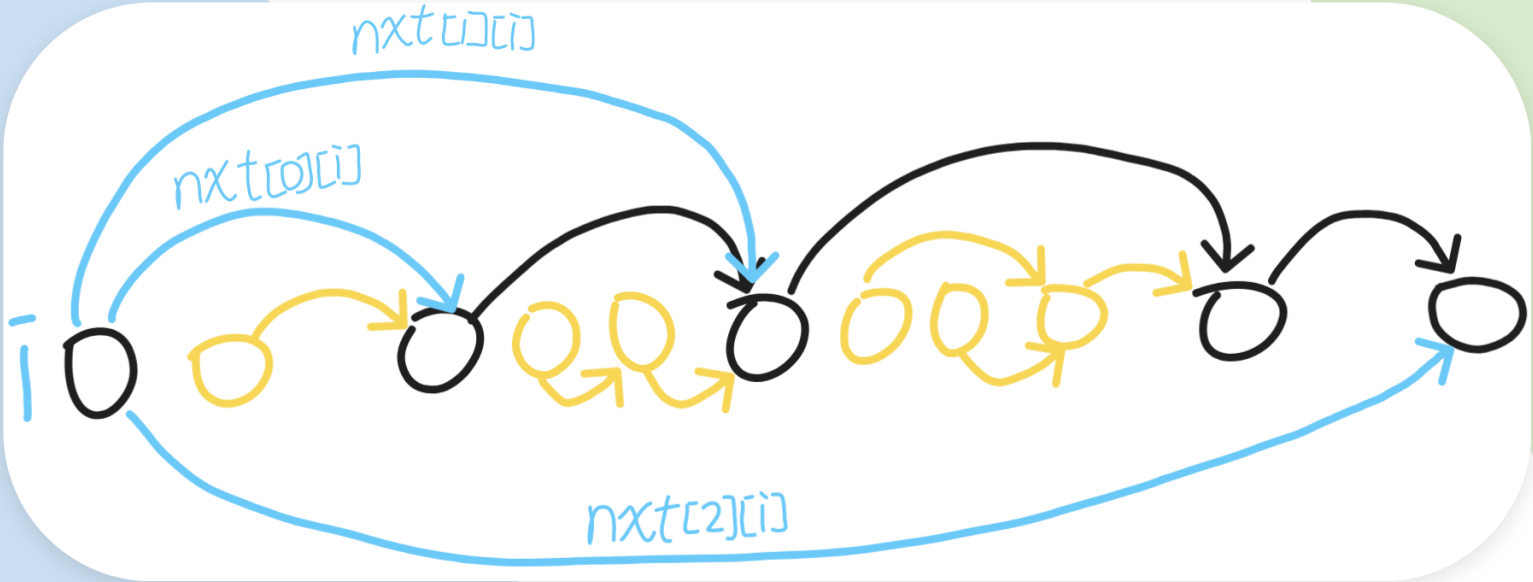
也使用倍增解决，先求出 $nxt[k][i]$ 表示从第 i 个盘子往后跳 2^k 步会跳到的盘子。



所求即，最少往后跳多少个圆盘能让跳过的圆盘容积之和不小于一开始水的体积。

也使用倍增解决，先求出 $nxt[k][i]$ 表示从第 i 个盘子往后跳 2^k 步会跳到的盘子。

再求出 $sum[k][i]$ 表示从第 i 个盘子起，以及往后跳 2^k-1 步这一共 2^k 个盘子的容积之和。

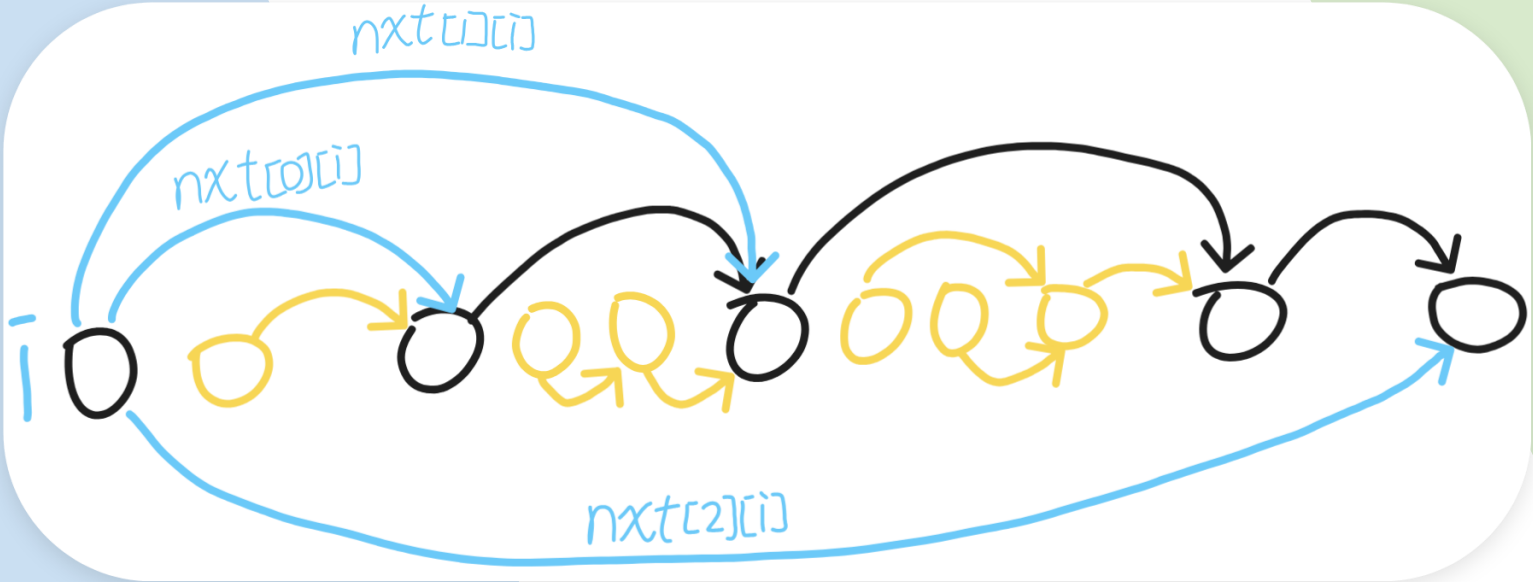


所求即，最少往后跳多少个圆盘能让跳过的圆盘容积之和不小于一开始水的体积。

$nxt[k][i]$ 表示从第 i 个盘子往后跳 2^k 步会跳到的盘子。

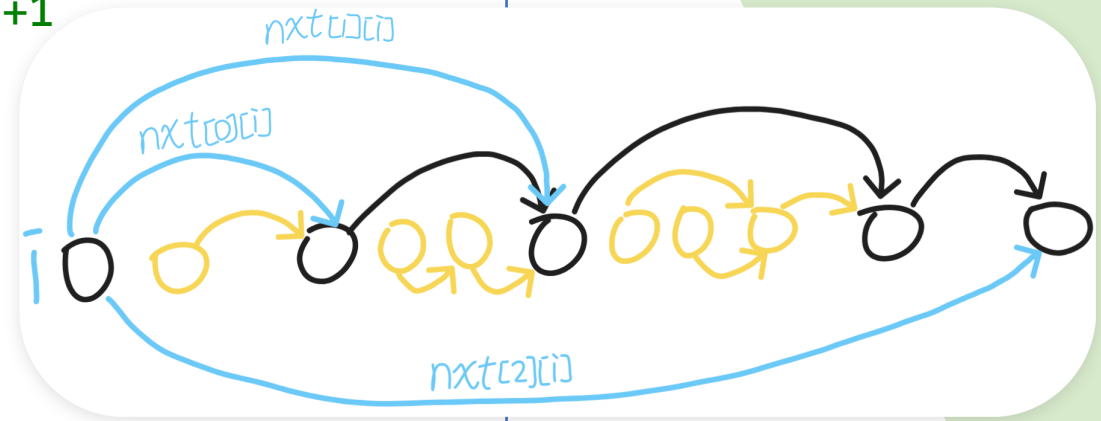
$sum[k][i]$ 表示从第 i 个盘子**起**，以及往后跳 2^k-1 步这一共 2^k 个盘子的容积之和。

如果体积为 x 的水会从第 i 个盘子流到第 $nxt[k][i]$ 个盘子里，说明 $x > sum[k][i]$。



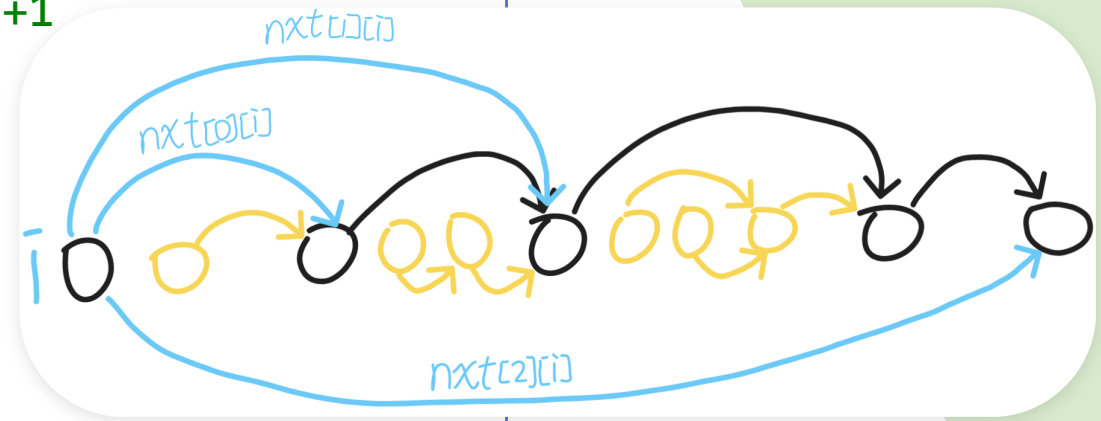
$nxt[k][i]$ 表示从第 i 个盘子往后跳 2^k 步会跳到的盘子。

```
for(int i=1;i<=N;i++)
{
    int pos=i+1;//[i,pos-1] <=R[i]
    for(int k=J;k>=0;k--)
    {
        if(pos+(1<<k)-1<=N&&mx[k][pos]<=R[i])
        //[pos,pos+2^k-1] <=R[i]
            pos+=(1<<k);
    }
//第一个大于自己的圆盘位置即为pos
//没有圆盘大于自己的时候pos=N+1
    nxt[0][i]=pos;
}
nxt[0][N+1]=N+1;
for(int k=1;k<=J;k++)
{
    nxt[k][N+1]=N+1;
    for(int i=1;i<=N;i++)
        nxt[k][i]=nxt[k-1][nxt[k-1][i]];
}
```



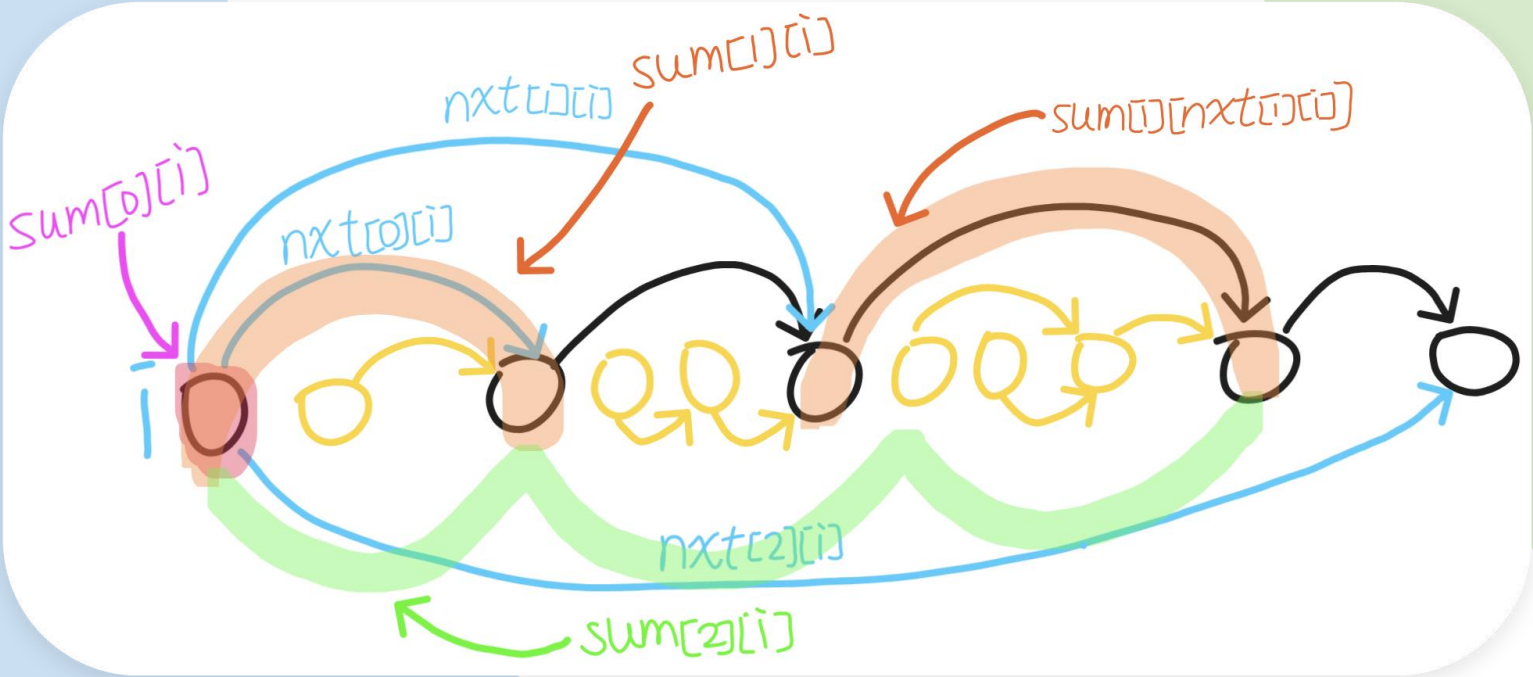
$nxt[k][i]$ 表示从第 i 个盘子往后跳 2^k 步会跳到的盘子。

```
for(int i=1;i<=N;i++)
{
    int pos=i+1;//[i,pos-1] <=R[i]
    for(int k=J;k>=0;k--)
    {
        if(pos+(1<<k)-1<=N&&mx[k][pos]<=R[i])
        //[pos,pos+2^k-1] <=R[i]
            pos+=(1<<k);
    }
//第一个大于自己的圆盘位置即为pos
//没有圆盘大于自己的时候pos=N+1
    nxt[0][i]=pos;
}
nxt[0][N+1]=N+1;
for(int k=1;k<=J;k++)
{
    nxt[k][N+1]=N+1;
    for(int i=1;i<=N;i++)
        nxt[k][i]=nxt[k-1][nxt[k-1][i]];
}
```



$sum[k][i]$ 表示从第 i 个盘子起，以及往后跳 2^k-1 步这一共 2^k 个盘子的容积之和。

```
for(int i=1;i<=N;i++)
    sum[0][i]=C[i];
for(int k=1;k<=J;k++)
    for(int i=1;i<=N;i++)
        sum[k][i]=sum[k-1][i]+sum[k-1][nxt[k-1][i]]
```



如果体积为 x 的水会从第 i 个盘子流到第 $nxt[k][i]$ 个盘子里，说明 $x > sum[k][i]$。

```
int r,v;
scanf("%d%d",&r,&v);
for(int k=J;k>=0;k--)
{
    if(nxt[k][r]>N) continue;
    if(sum[k][r]<v)
    {
        v-=sum[k][r];
        r=nxt[k][r];
    }
}
if(v<=C[r])
    printf("%d\n",r);
else printf("%d\n",0);
```

P8805 机房

P8805 [蓝桥杯 2022 国 B] 机房

P8805 机房

P8805 [蓝桥杯 2022 国 B] 机房

一台电脑的延迟就是与这台电脑相连的电脑数量。

电脑 u_i 到电脑 v_i 发送信息的最短时间就是从 u_i 走到 v_i 路径上所有电脑的延迟之和。

P8805 机房

使用树上倍增完成

P8805 机房

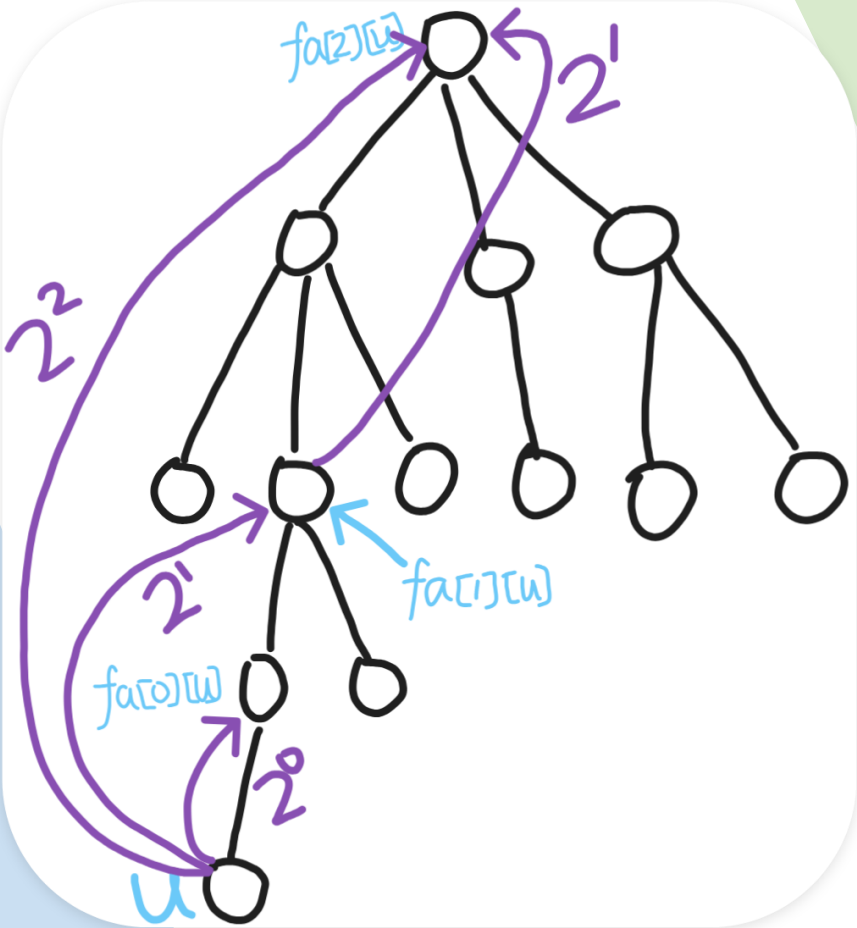
使用树上倍增完成

预处理 $fa[k][u]$ 表示节点 u 的第 2^k 级祖先节点。

预处理 $sum[k][u]$ 表示节点 u 以及其 2^k-1 级祖先中这一共 2^k 个点的延迟之和。

预处理 $fa[k][u]$ 表示节点 u 的第 2^k 级祖先节点。

$$fa[k][u] = fa[k-1][fa[k-1][u]]$$

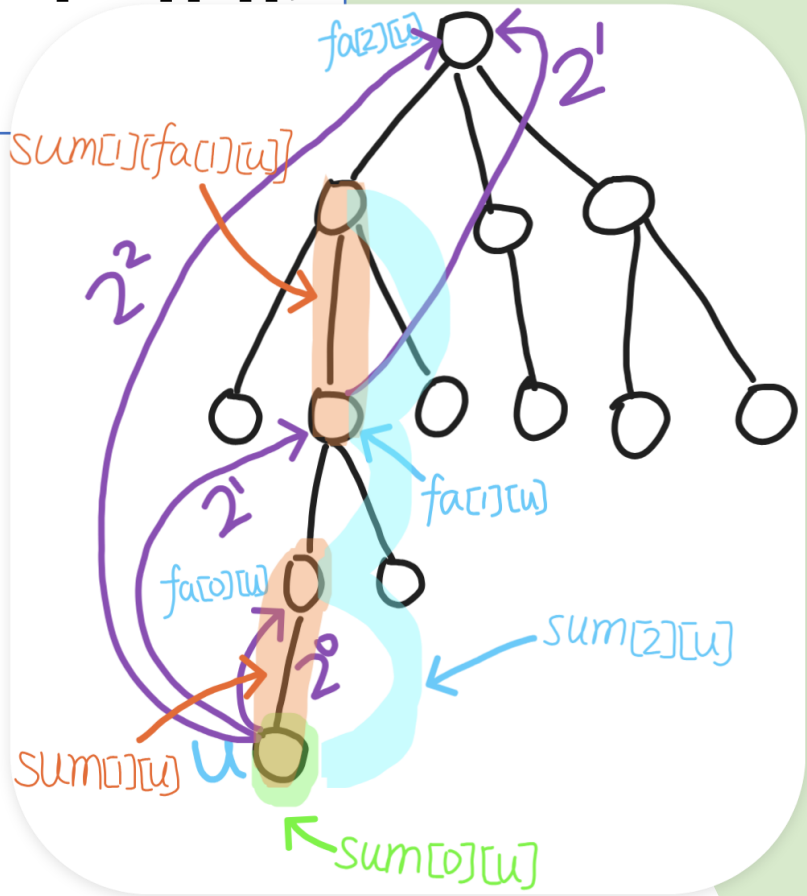


预处理 $fa[k][u]$ 表示节点 u 的第 2^k 级祖先节点。

预处理 $sum[k][u]$ 表示节点 u 以及其 2^k-1 级祖先中这一共 2^k 个点的延迟之和。

```
for(int k=1;k<=J;k++)
    for(int i=1;i<=n;i++)
    {
        sum[k][i]=sum[k-1][i]+sum[k-1][fa[k-1][i]];
        fa[k][i]=fa[k-1][fa[k-1][i]];
    }
```

$u_i \to v_i$ 拆分成 $u_i \to LCA + v_i \to LCA$



预处理 $fa[k][u]$ 表示节点 u 的第 2^k 级祖先节点。

预处理 $sum[k][u]$ 表示节点 u 以及其 2^k-1 级祖先中这一共 2^k 个点的延迟之和。

$u_i \to v_i$ 拆分成 $u_i \to LCA + v_i \to LCA$

```
int query(int x,int y)
{
    if(dep[x]<dep[y]) swap(x,y);
    int d=dep[x]-dep[y],res=0;
    for(int k=0;k<=J;k++)
    {
        if(d&(1<<k))
        {
            res+=sum[k][x];
            x=fa[k][x];
        }
    }
    if(x==y)
    {
        res+=a[x];
        return res;
    }
```

不包括 fa[k][x]

```
    for(int k=J;k>=0;k--)
    if(fa[k][x]!=fa[k][y])
    {
        res+=sum[k][x];
        res+=sum[k][y];
        x=fa[k][x];
        y=fa[k][y];
    }
    res+=sum[0][x];
    res+=sum[0][y];
    res+=a[fa[0][x]];
    return res;
}
```