倍增
doubly

山东省实验中学
宁华

引例：#422. 幂运算

- 输入三个整数 a, b, p，求 a^b mod p
- 样例输入：
- 2 3 3
- 样例输出：
- 2
- 数据范围：
- 0 ≤ a, b, p ≤ 10^9 且 p ≠ 0

- long long a,b,p,s;
- cin>>a>>b>>p;
- s=1%p;
- for(int i=1;i<=b;i++)
- 	s=s*a%p;
- cout<<s<<endl;

- 超时！如何优化？

倍增

- 一、理解倍增
- 二、经典应用

一、理解倍增

- 1.倍增概念
- 2.倍增的体验
- 3.倍增的数学基础
- 4.理解数学的倍增方法

1.倍增概念

- 倍增就是成倍的增长
- 例如： 数字 1->2->4->8->16->32->64->128->……
- 倍增是根据已经得到的信息，将考虑的范围扩大一倍，从而加速操作的一种思想

2.倍增的体验——神奇的倍增

- 体验1：
- 一份工作，共需要 30 天完成，
- 假如有两种薪资报酬：
- A：每天 1 万元。
- B：第一天 1 分钱，然后，后一天是前一天的 2 倍。
- 你会选择哪一种？

2.倍增的体验——神奇的倍增

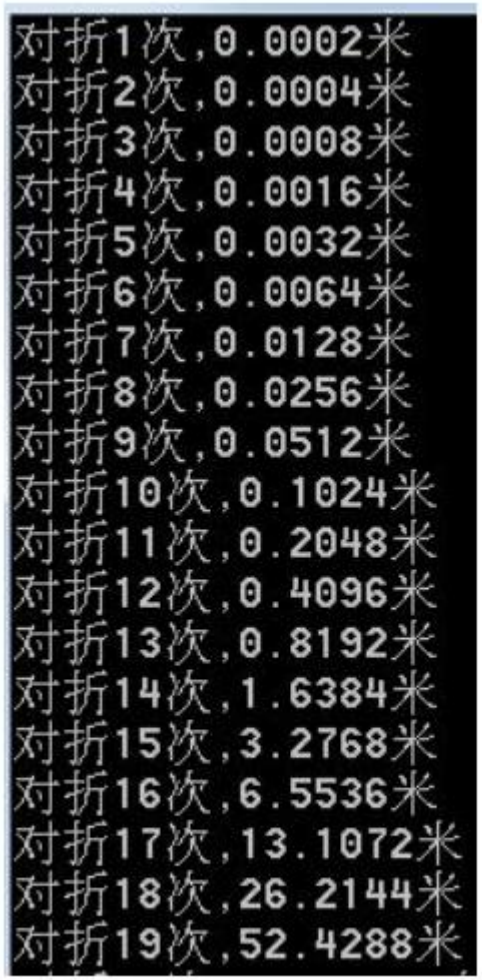
```
#include<cstdio>
int main()
{
    double sum=0, s=0.01;
    for(int i=1;i<=30;i++)
    {
        sum=sum+s;
        s=s*2;
    }
    printf("总钱数是%.2f元\n",sum) ;
    return 0;
}
```

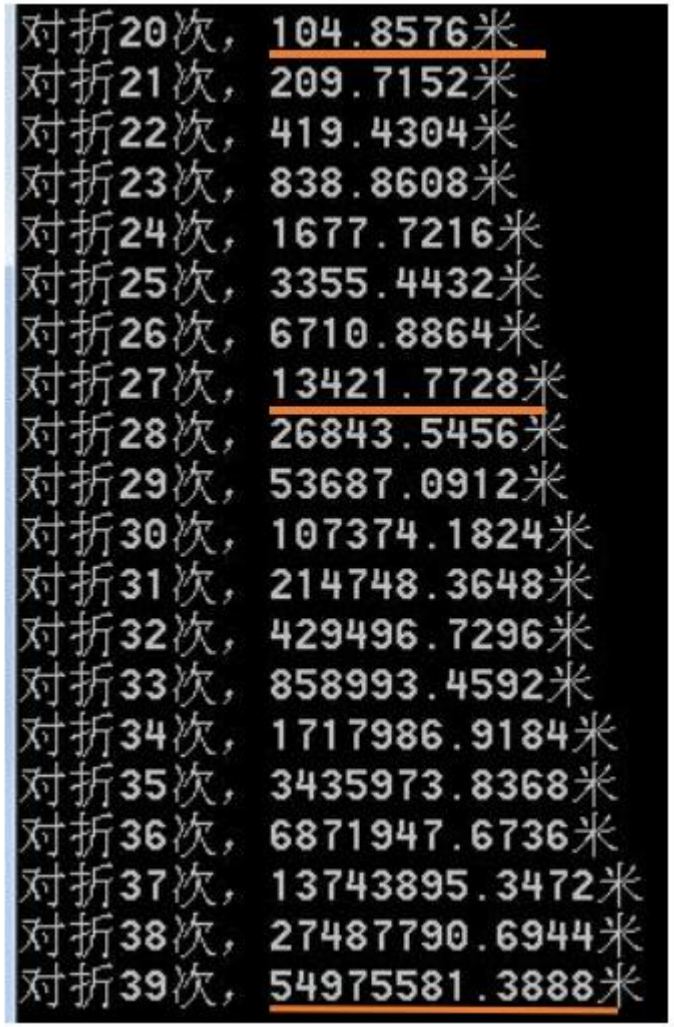
2.倍增的体验——神奇的倍增

- 体验2：
- 折纸万里：一张纸（0.1毫米厚），对折多少次可以超过百米高楼，珠穆朗玛峰，地球周长？
- 百米高楼 _____次
- 珠峰高度8848.86米 _____次
- 地球周长40076公里 _____次

2.倍增的体验——神奇的倍增

```
#include<cstdio>
int main()
{
    double h=0.1;
    double earth=40076.0*1000*1000;
    int i=0;
    while (h<earth)
    {
        i++;
        h=h*2;
        printf("对折%d次，%.4lf米\n",i,h/1000);
    }
    return 0;
}
```





2.倍增的体验——神奇的倍增

- 体验3：
- 这是一个很著名的故事：阿基米德与国王下国际象棋，国王输了，国王问阿基米德要什么奖赏？阿基米德对国王说：“我只要在棋
- 盘上第一格放一粒米，第二格放二粒，第三格放四粒，第四格放八粒…按这个方法放满整个棋盘就行．”国王以为要不了多少粮食，就随口答应了，结果~~~

- 1公斤的大米有 6×10^4粒左右

- 1 + 2 + 4 + ... + 2^63
- = 2^64 - 1
- = 1.84467×10^19 粒
- ≈ 3×10^14公斤

3.倍增的数学基础——幂运算

- 幂运算，就是同一个数值的连乘
- n 个 a 相乘：a×a×……×a 记为 a^n
- 读作“a的n次方”或者“a的n次幂”，其中a叫做底数，n叫做指数。

- $2^{10} \times 2^5 = 2^{15}$
- $(2^3)^2 = 2^6$
- $(2\times10)^3 = 2^3\times10^3$

4.理解数学的倍增方法

- ①倍增的数字：1->2->4->8->16->32->64->128 ->.....
- 代表 2^0 -> 2^1 -> 2^2 -> 2^3 -> 2^4 -> 2^5 -> 2^6 -> 2^7
- ②计算A^n
- 一个数A, 就是A^1
- 倍增一次就是A*A：$A^1*A^1=A^2$
- 再倍增一次，就是让A^2自乘：$A^2*A^2=A^4$
- 很容易看出，倍增7次就可以求出A^{128}

4.理解数学的倍增方法

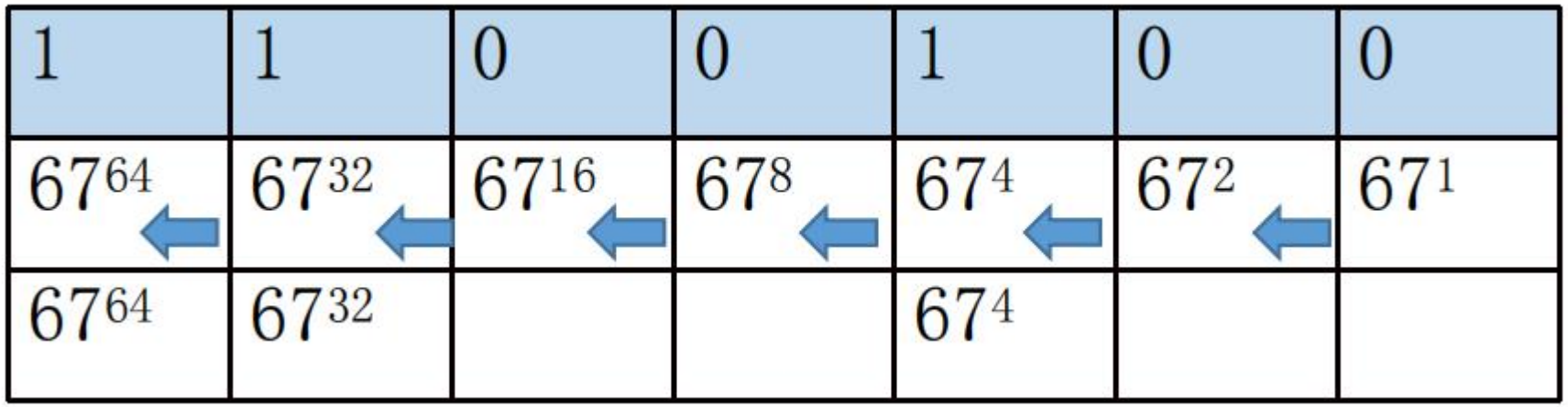
- 难度加大，如何求A^{100}？
- 先看100可以利用1,2,4,8……2^n某几个相加获得？
- 分析得：100=4+32+64

- 深入分析一下:
- $(100)_{10}=(1100100)_2$

7	6	5	4	3	2	1	0
128	64	32	16	8	4	2	1
	1	1	0	0	1	0	0
	64	32			4		

4.理解数学的倍增方法

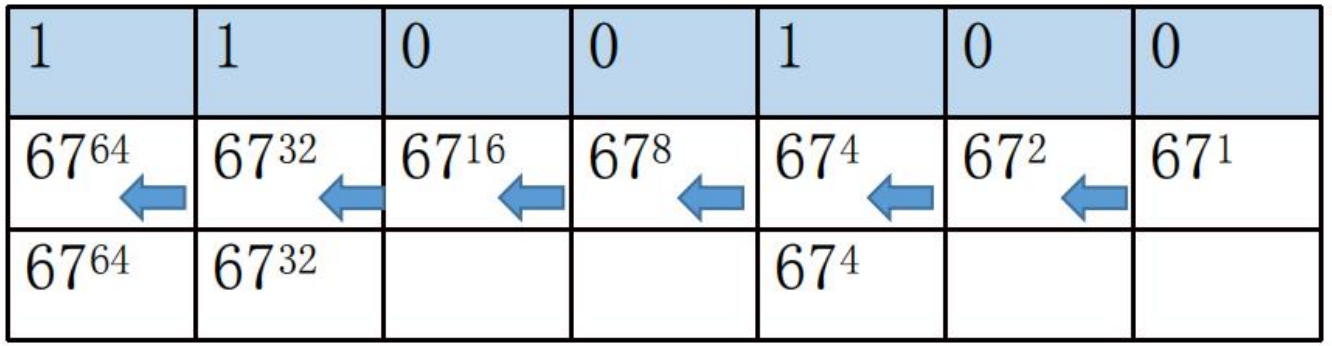
- 如果求67^{100}，根据幂运算规则



- A^n计算次数：n的二进制位数

求 a^b，如果不考虑溢出：

```
#define ll long long
ll ksm(ll a, ll b)
{
    ll ans=1;
    while(b)
    {
        if(b&1)ans=ans*a;
        b>>=1;
        a=a*a;
    }
    return ans;
}
```



二、经典应用

- 1.快速幂
- 2.龟速乘
- 3.RMQ问题的ST(Sparse Tabel)算法
- 4.LCA问题

1.快速幂 #422. 快速幂

```
#define ll long long
ll ksm(ll a, ll b, ll p)
{
    ll ans=1%p;
    a=a%p;
    while(b)
    {
        if(b&1)ans=ans*a%p;
        b>>=1;
        a=a*a%p;
    }
    return ans;
}
```

1	1	0	0	1	0	0
67^{64} ←	67^{32} ←	67^{16} ←	67^{8} ←	67^{4} ←	67^{2} ←	67^{1}
67^{64}	67^{32}			67^{4}		

特别注意：

1.全部定义为 long long

2.
(1) ans = 1%p;
(2) a = a%p;
(3) ans = ans*a%p; 不要缩写 ans*=a%p

课后习题

- #795. [NOIP2013提高组] 转圈游戏

2.龟速乘 #801. 龟速乘

- 题目描述
- 输入 a, b, p, 求 a × b mod p
- 样例输入
- 2 3 4
- 样例输出
- 2
- 数据范围
- 1 ≤ a, b, p ≤ 10^18 且 p ≠ 0

【算法分析】

- a*b 会超出 long long，不使用高精度的话，必须转为多次加运算。
- 与快速幂原理相同，将 b 转为二进制，利用乘法分配律求解。
- 例如 a=20, b=13
- 20*13
- = $20*(1101)_2$
- = $20*((2^3)*1 +(2^2)*1 + (2^1)*0 + (2^0)*1)$
- = $20*(2^3)*1 + 20*(2^2)*1 + 20*(2^1)*0 + 20*(2^0)*1$
- = 20*8*1 + 20*4*1 + 20*2*0 +20*1*1
- = 260
- 从右到左模拟此过程，循环取b的每一位，a每次循环均 *2，直到取完b

```
#define ll long long
inline ll mult_mod(ll a,ll b,ll p) //计算a*b mod p
{
    ll res=0;
    while(b)
    {
        if(b&1) res=(res+a)%p;
        a=(a*2)%p;
        b>>=1;
    }
    return res;
}
```

$$
\begin{aligned}
&20*13\\
&= 20*(1101)_2\\
&= 20*((2^3)*1 +(2^2)*1 + (2^1)*0 + (2^0)*1)\\
&= 20*(2^3)*1 + 20*(2^2)*1 + 20*(2^1)*0 + 20*(2^0)*1\\
&= 20*8*1 + 20*4*1 + 20*2*0 +20*1*1\\
&= 260
\end{aligned}
$$

3.ST(Sparse Tabel)算法

- 例 #802. 天才的记忆

- 给定一个长度为N的数列A，然后有Q次询问，每次询问给出区间L和R，回答A[L]~A[R]中最大值。(N,Q<=10^5)

【算法分析】

- **区间最值查询**(RMQ-Range Minimum/Maximum Query)

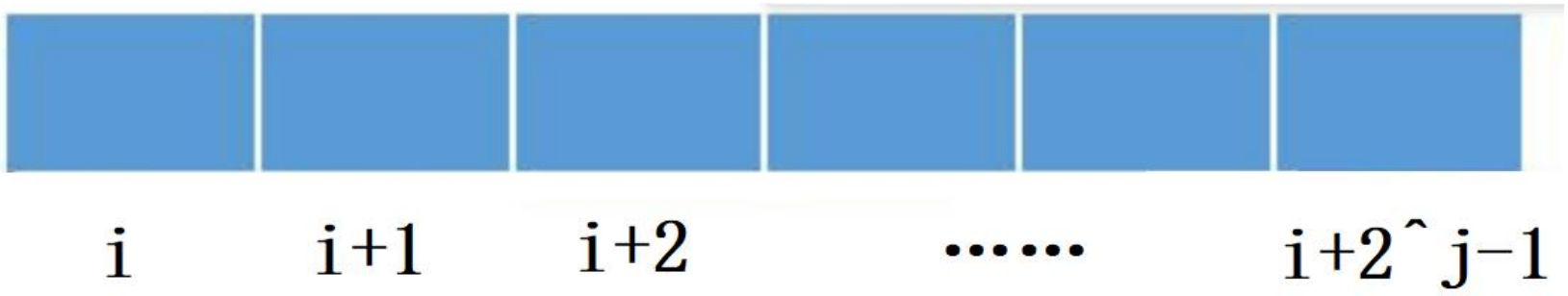
- 容易想到暴力查询，但当数据量非常大且查询很频繁时，查询效率低下。

ST算法

- Sparse Tabel 稀疏表——一个非常有名的在线处理RMQ问题的算法，编码简单，查询效率高。
- 所谓在线算法，是指用户每输入一个查询便马上处理一个查询。该种算法一般用较长的时间做预处理，待信息充足以后便可以用较少的时间回答每个查询。
- **ST算法前提：**数列是静态的
- **ST算法效率：**先预处理O(NlogN)，再查询（单次查询O(1)）
- 当然，该问题也可以用线段树（也叫区间树）解决，这里我们暂不介绍，感兴趣的同学课下自学。

（一）首先是预处理，构造稀疏表，用动态规划（DP）解决

- DP状态 f[i][j]：表示从第 **i** 个数起连续 **2^j** 个数中的最大值
- 即：以 i 为起点，长度为 2^j 的区间最值
- 区间是[i, i+2^j-1]



- 例如：
- A数列为：

1	2	3	4	5	6	7	8	9	10
3	2	4	5	6	8	1	2	9	7

- F[1,0]表示第1个数起，长度为2^0=1的最大值，其实就是3这个数。同理 F[1,1] = max(3,2) = 3, F[1,2]=max(3,2,4,5) = 5，F[1,3] = max(3,2,4,5,6,8,1,2) = 8, F[2,3]=max(2,4,5,6,8,1,2,9) = 9;
- 并且我们可以容易的看出F[i,0]就等于A[i]。（DP的初始值）
- 这样，DP的状态、初值都已经有了，剩下的就是状态转移方程。

分治思想

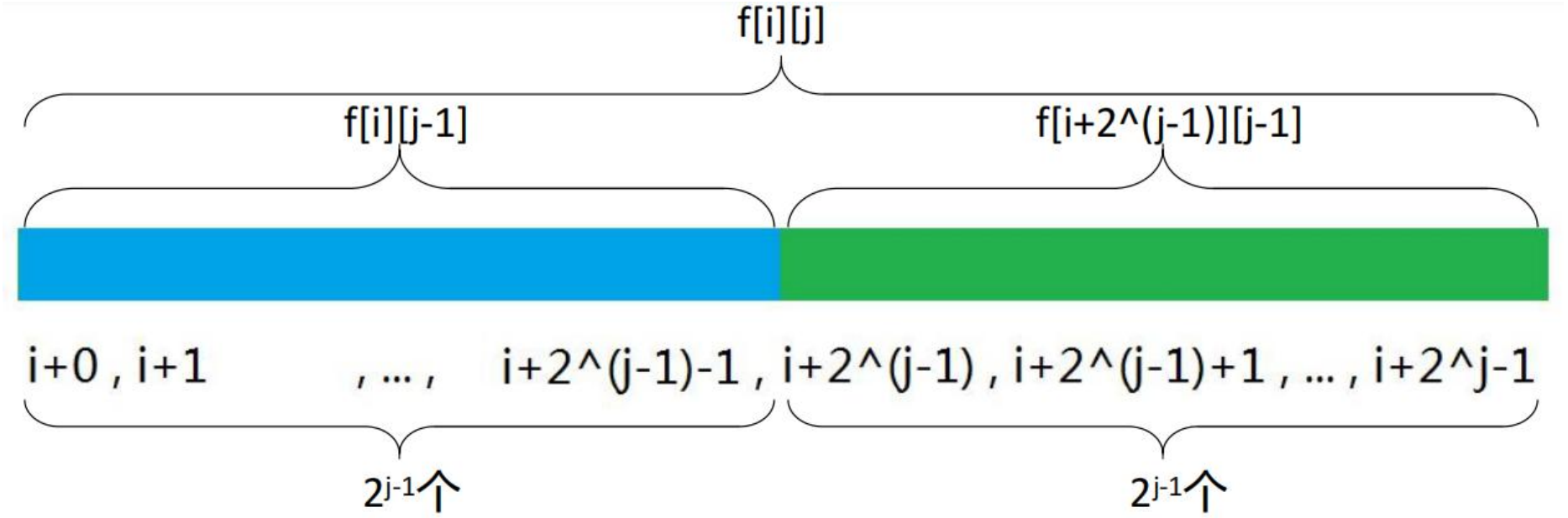
- 任何一个 f[i][j] 所表示的区间长度都是2的整数次方，即2^j，我们可以将该区间从中间断开，平均分为两段。
- 两段区间的起点分别是：i 和 i+2^(j-1)，每段的长度都是2^(j-1)

1	2	3	4	5	6	7	8	9	10
3	2	4	5	6	8	1	2	9	7

- 用上例说明，当 i=2，j=3 时就是2,4,5,6 和 8,1,2,9 这两段。

分治思想

- 于是，可以得到： f[i][j]=max(f[i][j-1], f[i + 2^(j-1)][j-1]) （以求区间最大值为例）



递推实现（dp）

- f[i][0]=a[i]
- f[i][j]=max(f[i][j-1],f[i+(1<<(j-1))][j-1])

预处理

- void INIT_RMQ()//预处理->O(nlogn)
- {
- for(int i=1; i<=n; i++)
- f[i][0]=a[i]; //边界，长度为1的区间
- for(j=1; (1<<j)<=n; j++) // 区间长度
- for(int i=1; i+(1<<j)-1<=n; i++) //对每个起始点，都可以找到 n，所以i+2^j-1<=n
- f[i][j]=max(f[i][j-1],f[i+1<<(j-1)][j-1])
- }
- 说明：
- ①a[]:存原始数据，可省，直接 cin>>f[i][0]。
- ②f[n][m]数组大小，n为数据元素的个数，m为log2(n)=log(n)/log(2)
- ③区间长度由小到大枚举，利用区间动规的思想，
- ④区间最值得求解利用了倍增的思想
- ⑤算法时间复杂度为O(n log n)

可能常见其他写法

```
void INIT_RMQ()//预处理->O(nlogn)
{
    for(int i=1;i<=n;i++)
        F[i][0]=A[i];
    int m=log2(n);
    for(int j=1;j<=m;j++)
        for(int i=1;i<=n-(1<<j)+1;++i)
            F[i][j]=max(F[i][j-1],F[i+(1<<(j-1))][j-1]);
}
```

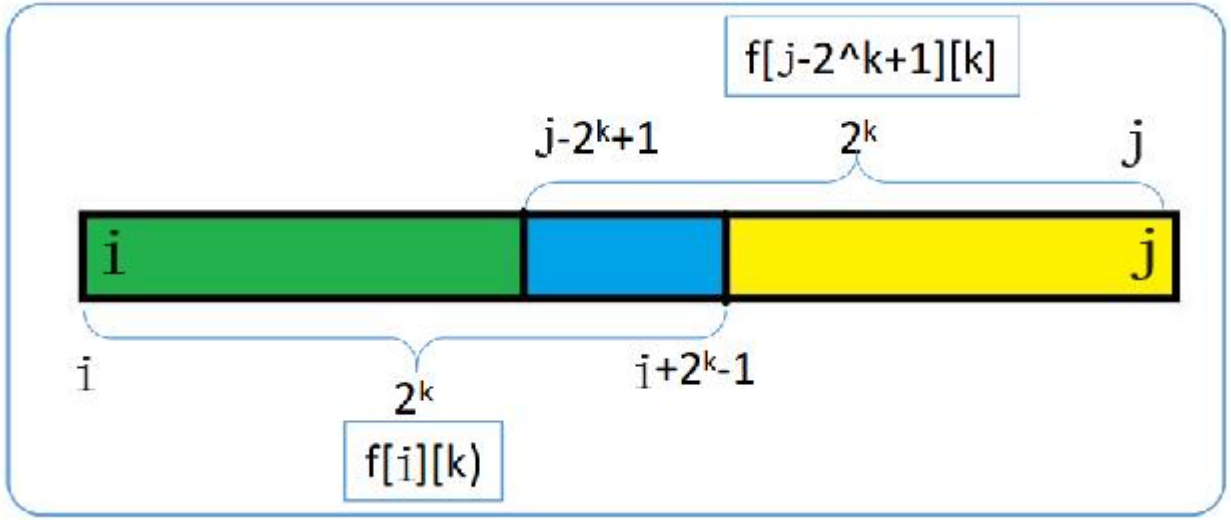
- 不论哪种写法，这里我们需要注意的是循环的顺序：我们发现外层是j，内层是i，这是为什么呢？可以是i在外，j在内吗？

- 答案是不可以。因为我们需要理解这个状态转移方程的意义。
- 状态转移方程的含义是：先初始化所有F[i,0]即1个元素，然后通过两个1个元素的最值，获得所有F[i,1]即2个元素的最值，然后再通过两个2个元素的最值，获得所有F[i,2]即4个元素的最值，以此类推依次更新所有长度的最值。
- 而如果是i在外，j在内的话，我们更新的顺序就是F[1,0],F[1,1],F[1,2],F[1,3],表示更新从A[1]开始1个元素的，2个元素的，4个元素的，8个元素(A[1],....A[8])的最值。例如 F[1,3] = max(F[1,2],F[5,2])
- = max(max(A[1],A[2],A[3],A[4]),max(A[5],A[6],A[7],A[8]))的值，但是我们根本没有计算过 F[5,2]，所以这样的转移肯定是错误的。
- 为了避免这样的错误，一定要**好好理解这个状态转移方程所代表的含义**。

（二）然后是查询。

- 如何回答每次查询呢？
- 对查询区间 [i,j]，其长度为 j-i+1，
- 由于区间的长度很可能不是 2 的整数幂，所以我们要把区间划分为长度为 2的整数幂 的两部分，而且这两个区间的并集必须是 [i,j]。
- 找满足2^k <= (j-i+1) 的最大k，把区间分成两部分[i,i+2^k-1]和 [j-2^k+1,j]

- $k = \lfloor \log_2(j-i+1) \rfloor$



比如数列： 3 5 2 4 7 1 2 9 6 , 共9个数

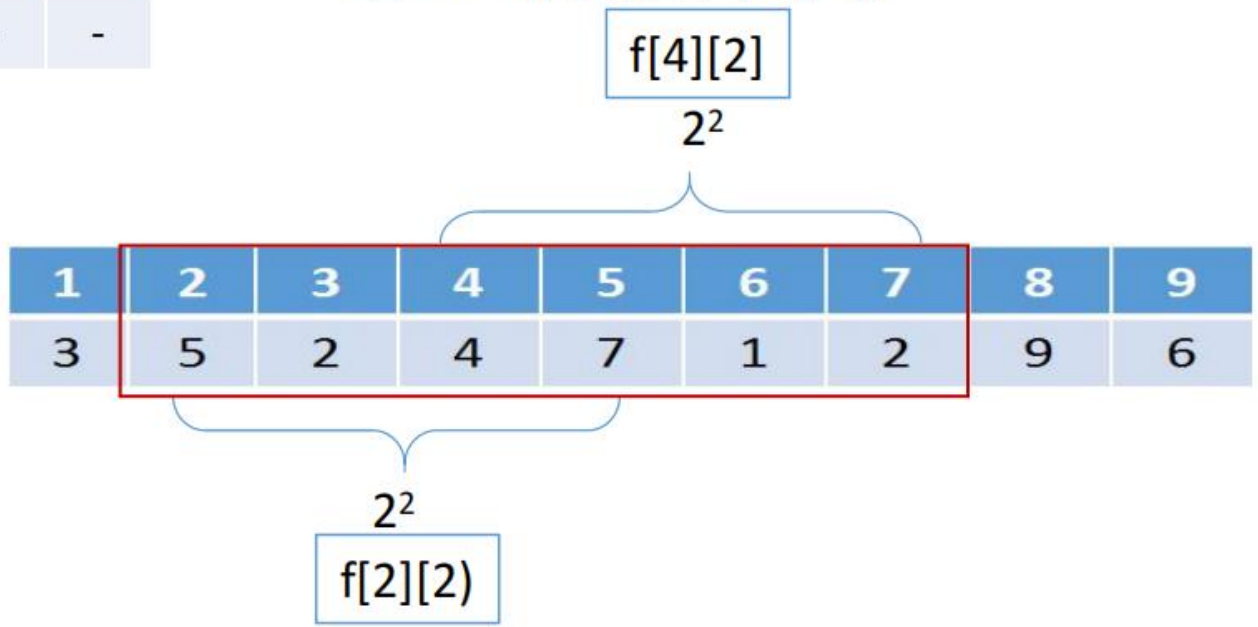
F	1	2	3	4	5	6	7	8	9
0(1)	3	5	2	4	7	1	2	9	6
1(2)	5	5	4	7	7	2	9	9	-
2(4)	5	7	7	7	9	9	-	-	-
3(8)	9	9	-	-	-	-	-	-	-

询问区间[2,7]的最大值？

① 区间长度为7-2+1=6

② $k = \lfloor \log_2 6 \rfloor = 2$

③ max(f[2][2],f[4][2])



- int RMQ(int i,int j)
- {
- 	int k=log2(j-i+1);//int k=(int) (log(j - i + 1) / log(2));
- 	return max(F[i][k],F[j-(1<<k)+1][k]);
- }
- **在这里我们也需要注意一个地方，就是<<运算符和+ -运算符的优先级。**
- k也可以通过循环求出
- int k = 0;
- while((1<<k) <= y-x+1) k++;
- k--;

模板题：poj3264

- **Description**
- 在每天挤奶的时候，农民约翰的N头牛（1≤n≤50000）总是排成一列。有一天，约翰决定与他的牛们一起玩一个极限飞盘游戏。为了简单起见，他将从奶牛队列里面选一定范围内的奶牛来玩这个游戏。然而所有的牛对这个游戏都很感兴趣。农民约翰列出了Q份名单（1≤Q≤200000）和每个奶牛的高度（1≤高度≤1000000）。对于每一份名单，他想你帮助他确定在每份名单中高度最高的奶牛与高度最低的奶牛的高度差是多少。
- **Input**
- 第一行为N(1≤N≤50000)和Q(1≤Q≤200000)；从第2行到第N+1行，每行一个数字，表示第i头牛的高度(1≤height≤1000000)；从第N+2行到第N+Q+1行，每行两个整数A和B(1≤A≤B≤N)，表示从第A头牛到第B头牛的范围。
- **Output**
- 从第一行到第Q行，每行一个整数，表示从第A头牛到第B头牛之间，最高牛与最矮牛的高度差。

- **Sample Input**
- **6 3**
- **1**
- **7**
- **3**
- **4**
- **2**
- **5**
- **1 5**
- **4 6**
- **2 2**
- **Sample Output**
- **6**
- **3**
- **0**

练习题：poj3368

- **Description**
- 给你一个含有n个整数的非递减序列a1 , a2 , ... , an，要求你回答一系列的询问，如i和j (1 ≤ i ≤ j ≤ n)，回答出ai , ... , aj之间出现频率最多数字为多少次？
- **Input**
- 输入文件包含多组数据，每组数据的第1行包含两个整数n和q(1 ≤ n, q ≤ 100000)，第2行为n个用空格分开的整数a1 , ... , an (-100000 ≤ ai ≤ 100000, for each i ∈ {1, ..., n})，接下来的q行，每行包含两个整数i和j。文件以0作为结束标记。
- **Output**
- 对于每一个询问输出一行，为在此区间内出现频率最多数字为多少次？

- Sample Input
- 10 3
- -1 -1 1 1 1 1 3 10 10 10
- 2 3
- 1 10
- 5 10
- 0
- Sample Output
- 1
- 4
- 3
- **Hint**
- 【数据范围】
  对于30的数据，1≤n,q≤1000；
  对于100的数据，1≤n,q≤1000000;

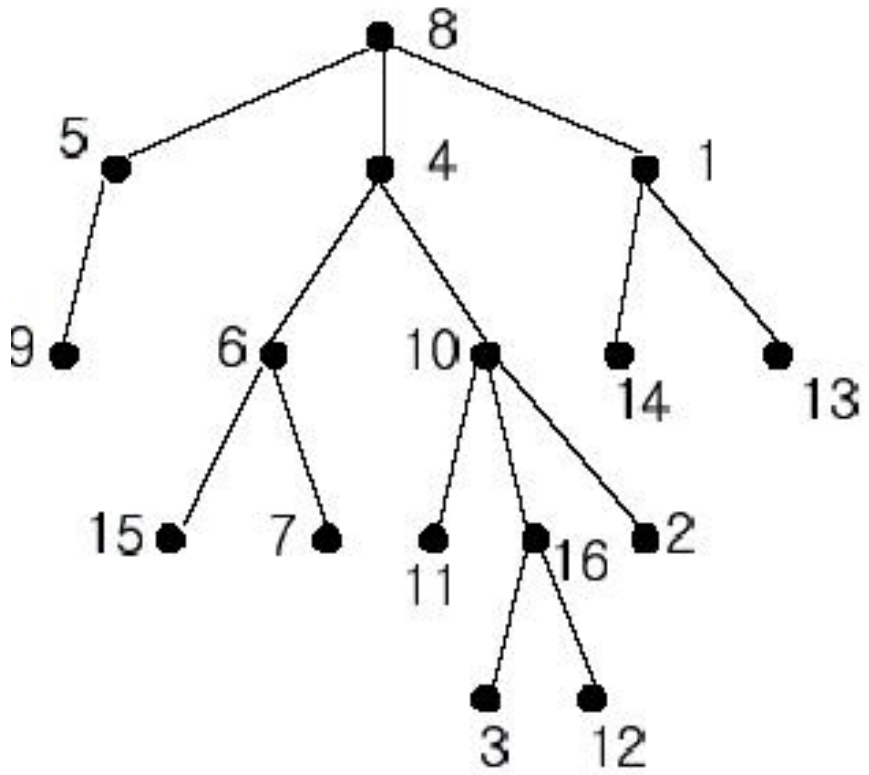
课后拓展提高

- 二维RMQ
- http://poj.org/problem?id=2019
- 给你一个n*n的矩阵，让你从中圈定一个小矩阵，其大小为b*b，有k个询问，每次询问告诉你小矩阵的左上角，求小矩阵内的最大值和最小值的差。
- Sample Input
- 5 3 1
- 5 1 2 6 3
- 1 3 5 2 7
- 7 2 4 6 1
- 9 9 8 6 5
- 0 6 9 3 9
- 1 2
- Sample Output
- 5

4.最近公共祖先

- LCA Least Common Ancestors

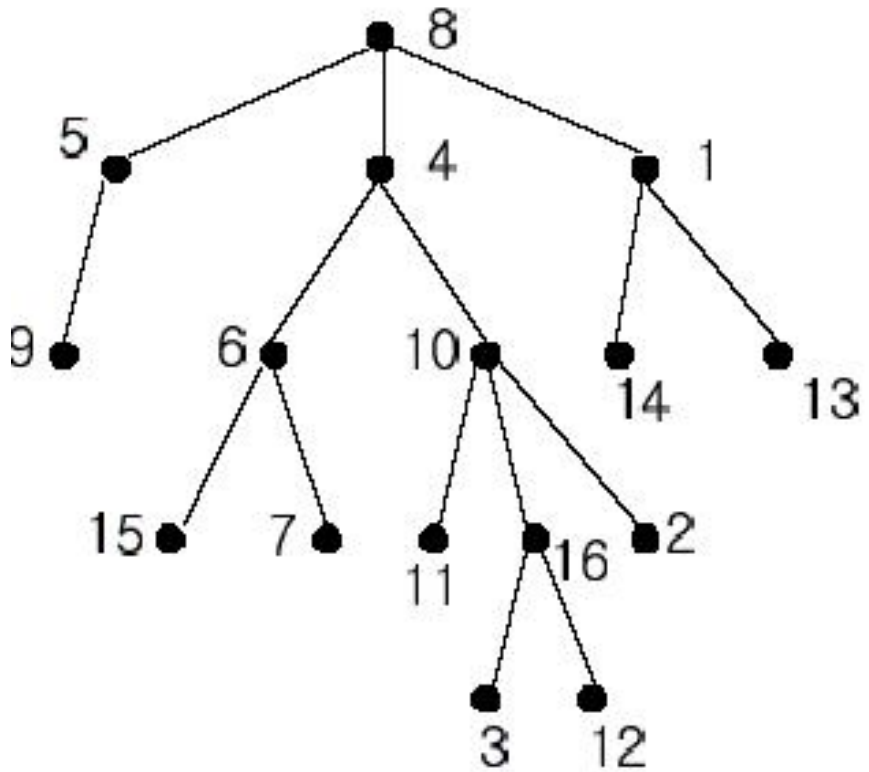
模板题POJ1330



LCA

- 一、暴力算法
- 二、在线算法——ST算法
- 三、倍增法
- 四、离线算法——Tarjan算法

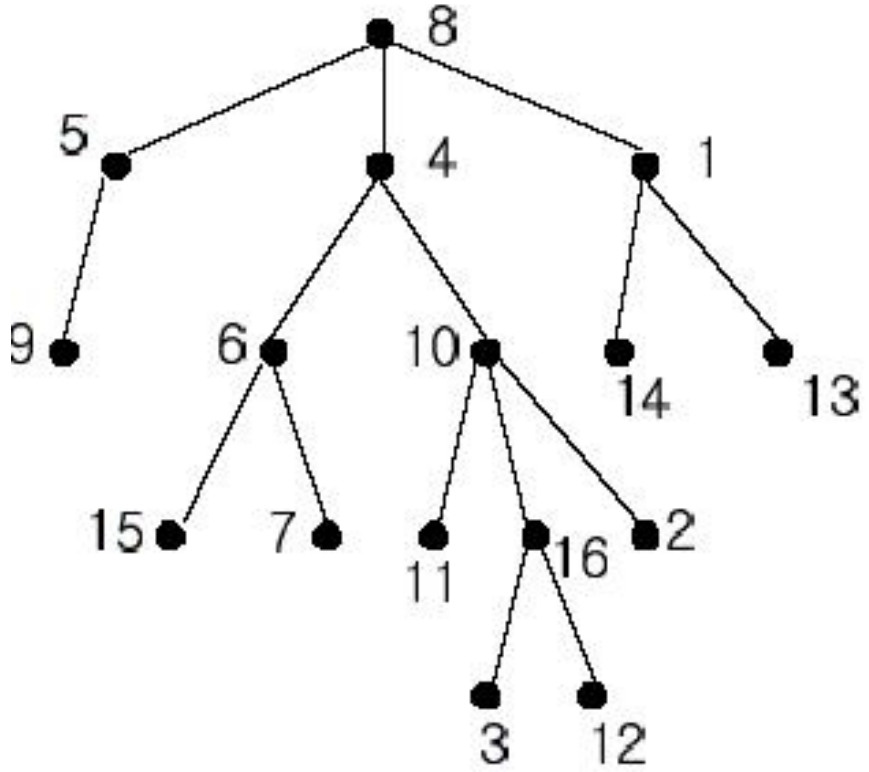
一、暴力枚举（朴素算法）



参考程序

- POJ1330-1.cpp
- 缺点？

改进？



- 改进：
- 先遍历求出每个结点的父亲和深度
- 对于查询LCA(u,v)，用p1、p2指向u、v，将p1、p2中深度较大的结点不断指向其父结点，直到p1、p2深度相同。
- 之后p1、p2同步向上移动，直到p1=p2，此时p1、p2所指向的结点就是LCA(u,v)。

参考程序

- POJ1330-2.cpp

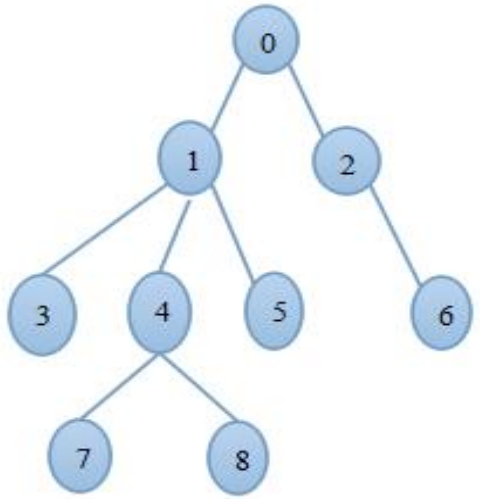
二、ST算法

- 在线算法：每次读入一个查询，处理这个查询，给出答案。
- 离线算法：一次性读入所有查询，统一进行处理，给出所有答案。

- 它们解决的问题都是询问式的，但是方法和特点不同，而且适用范围也不同(如果询问给出是有间隔的，往往只能用在线算法)。

ST算法

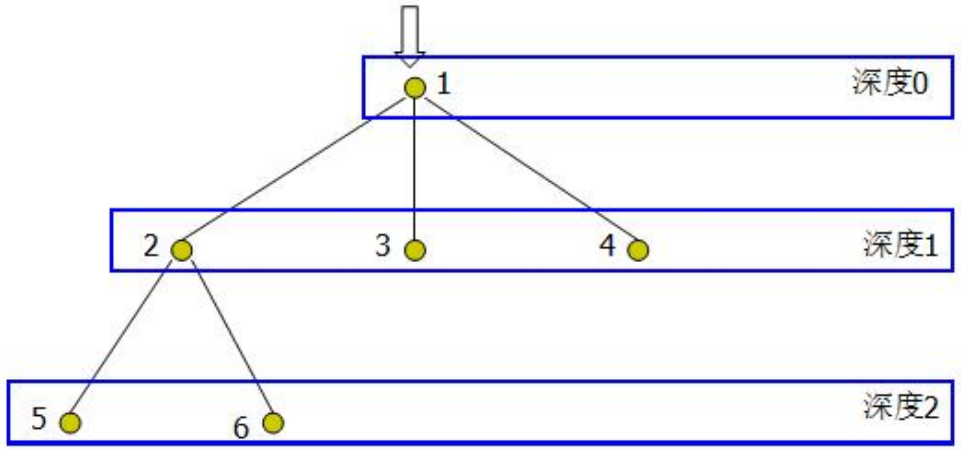
- **原理：树上任意两点的最近祖先，必定就是这两个节点的最短路径上深度最小的那个点。**
- 例如：下图中，节点7和5，其最短路径为7--4--1--5, 这条路径上深度最小的点为节点1，其深度为1。节点1即为节点7和5的LCA。



- 因此，要找到任意两个节点的LCA，只需要先找到上述最短路径，再找到最短路径中深度最小的点。而这就是下面所述LCA在线算法所做的事。

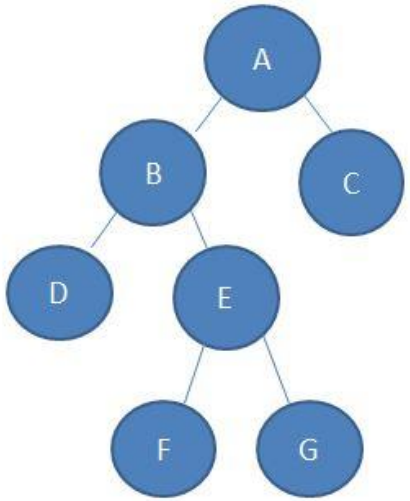
ST算法

- Euler序长度为2n-1
- 记录节点u在欧拉序列中第一次出现的位置为pos(u)



欧拉序列F：	1	2	5	2	6	2	1	3	1	4	1
深度序列B：	0	1	2	1	2	1	0	1	0	1	0
Pos (u)：	1	2	8	10	3	5					

遍历，结果保存在数组中



数组下标:	1	2	3	4	5	6	7	8	9	10	11	12	13
遍历序列：	A	B	D	B	E	F	E	G	E	B	A	C	A
节点在树中的深度:	1	2	3	2	3	4	3	4	3	2	1	2	1

要查询D和G的LCA：

1.在遍历序列中找到D和G第一次出现的位置，first[D] = 3 ， first[G] = 8 (3,8指数组下标)

2.取节点数组的[3,8]的那段序列，查询一个最小值

<3 , 2 , 3 , 4 , 3 , 4>，最小值为2，对应的节点是B，所以D,G的LCA是B

3.用ST算法可以查询到最小值，但是仅仅是知道最小值的值是多少，并不知道最小值在哪

ST算法本质是DP，在预处理构建DP数组的时候，再用另一个数组pos数组记录位置

dp[i][j] ： 表示从下标i开始长度为2^j的那段序列中的最小值

pos[i][j]： 表示对应dp[i][j]状态的那个最小值的数组下标

例如<2,1,2,3,4,5> dp[1][2]表示从下标1开始到下标4里面的最小值

dp[1][2] = 1 , pos[1][2] = 2 ， 表示1在下标2的位置

- LCA在线算法其实就是将LCA问题转化成RMQ问题：

- 1．对有根树T进行DFS，将遍历到的结点按照顺序记下，我们将得到一个长度为2N-1的序列，称之为T的欧拉序列F。
- 2．每个结点都在欧拉序列中出现，我们记录结点u在欧拉序列中第一次出现的位置为pos(u)。
- 3. 根据DFS的性质，对于两结点u、v，从pos(u)遍历到pos(v)的过程中经过LCA(u, v)有且仅有一次，且深度是深度序列B[pos(u)…pos(v)]中最小的，然后就转化成RMQ！
  LCA(T, u, v) → RMQ(B, pos(u), pos(v))

三、倍增LCA

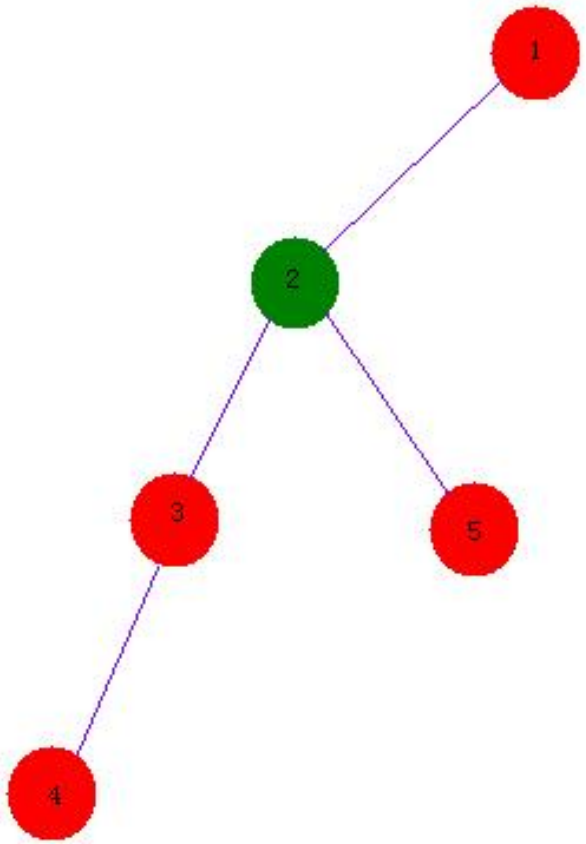
- 与RMQ的ST算法类似，我们令F[i][k]为结点i的第2^k个祖先结点。
- 则F[i][0]为i的第2^0=1个祖先结点，即父结点。令w为i的第2^(k-1)个祖先结点即w=F[i][k-1]，那么w的第2^(k-1)个祖先结点就是i的第2^k个祖先结点即F[i][k]=F[w][k-1]。

- 在查询LCA时，与朴素LCA类似，先将深度较大的结点u提升到与v的深度相同，而这一次我们利用**倍增法，一次提升2^k个祖先结点，**加快了算法的效率。

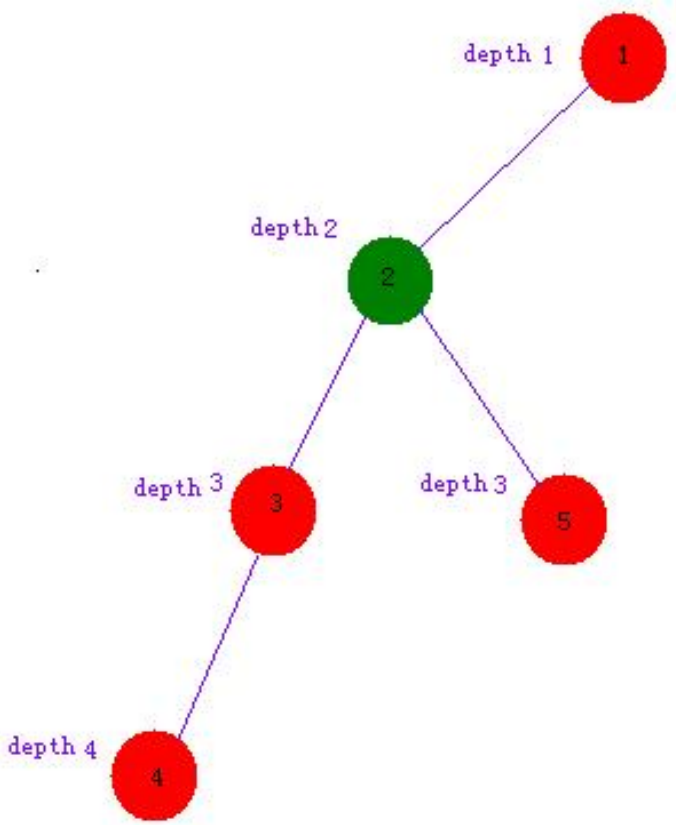
- 如果此时u和v相同，则u就是lca。否则，两个结点同时提高2^k(k是使2^k<=dep[u]最大的正整数)进行尝试，若二者碰头了，则减半步幅(k--)，直到u、v**到达不同的一个结点才跳**。当k减到0时，那么**所在结点的父亲**就是LCA(u,v)。

- **倍增法的优点在于，除了能求出LCA(u,v)，还可以对树上的路径进行维护。**
- 例如要求出结点u到结点v路径上最大的边权w，我们可以在预处理F[i][k]时，用一个数组maxCost[i][k]记录结点i到它的第2^k个祖先结点的路径上最大的边权。
- 那么在查询LCA(u,v)的过程中，求出u、v到公共祖先的路径上的最大边权，即u到v的路径上的最大边权。

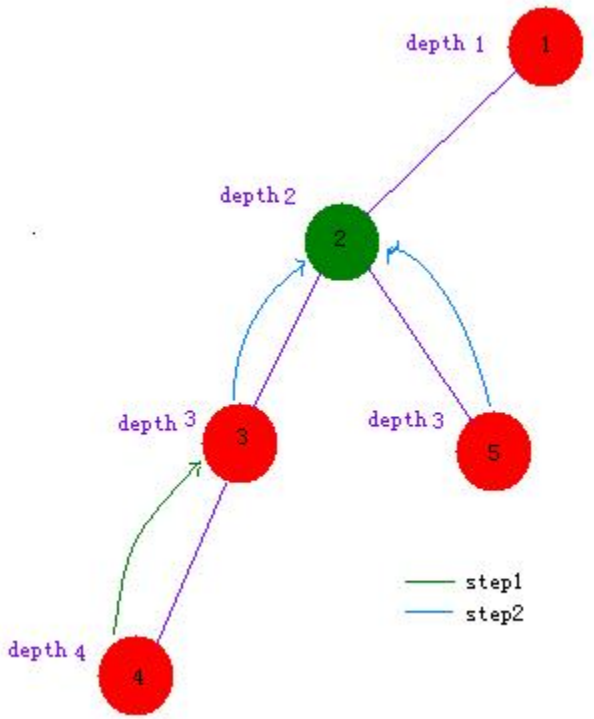
- 如图:
- 4和5的LCA就是2
- 怎么求呢？



- 最粗暴的方法就是先dfs一次，处理出每个点的深度



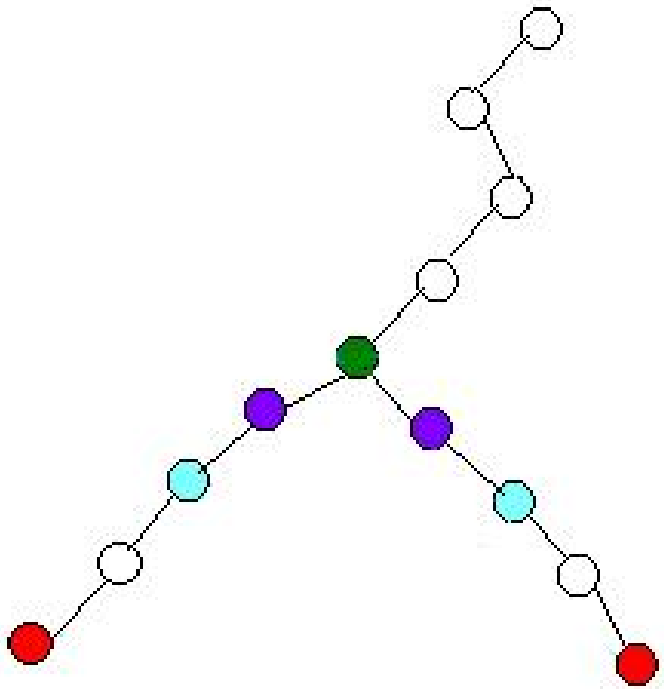
- 然后把深度更深的那一个点（4）一个点地一个点地往上跳，直到到某个点（3）和另外那个点（5）的深度一样
- 然后两个点一起一个点地一个点地往上跳，直到到某个点（就是最近公共祖先）两个点相遇“变”成了一个点



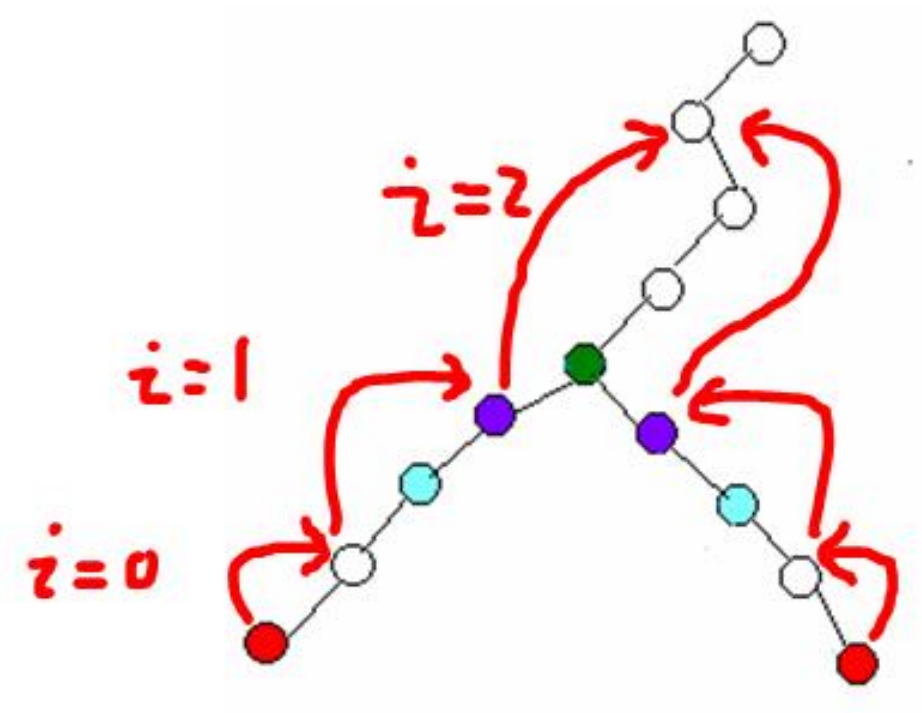
- 不过有没有发现一个点地一个点地跳很浪费时间？
- 如果一下子跳到目标点？又可能不好实现或者无法实现，相对来说倍增的性价比算是很高的。

- 倍增的话就是一次跳2^i 个点，不难发现深度差为x时，深度更深的那个点就需要跳x个点
- 于是可以写出这段代码

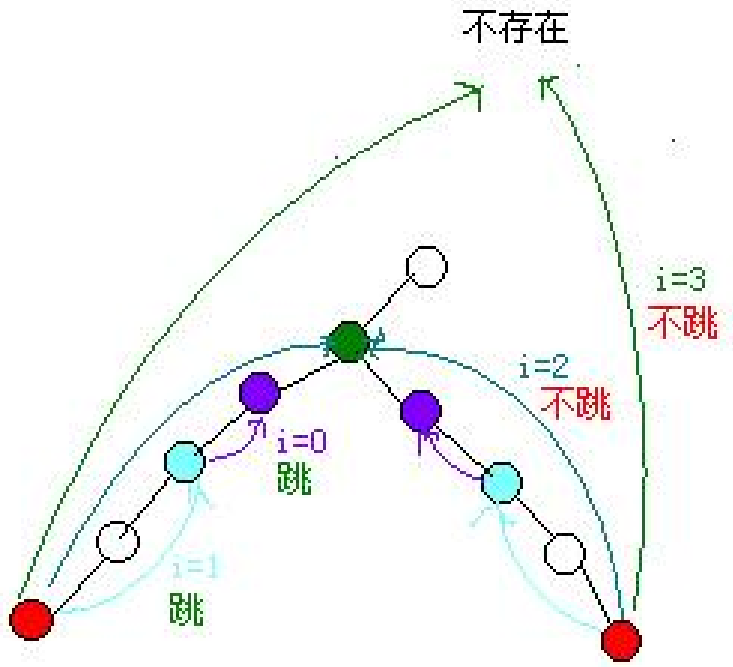
- if(depth[a] < depth[b])swap(a, b);
- int d = depth[a] - depth[b];
- for(int i = 0; i <= H; i++)
- 	if(d & (1 << i))
- 	a = f[a][i];　//f[a][i]表示a的第2^i个祖先结点
- 这样a和b就位于同一深度了，接下来怎么继续往上跳呢？



- 接下来很快就会发现一个很严重的问题:
- 两个点按照这样跳，不能保证一定是最近的



- 所以倍增找lca的方法是这样的:
- 从最大可以跳的步数开始跳(一定是2^i),
- 如果跳到的位置一样，就不跳
- 如果不一样才跳，每次跳的路程是前一次的一半



- 过程大概就像上图所示，但是执行完了最后一跳所到的点不是最近公共祖先，但是，它们再往上跳一个，就到了。

1. DFS预处理出所有节点的深度和父节点

```
void dfs(int u, int fu)
{
    f[u][0]=fu;
    depth[u]=depth[fu]+1;
    for(int i=h[u];i;i=e[i].nxt)
    {
        int v=e[i].to;
        if(v==fu)continue;
        dfs(v,u);
    }
}
```

2. 初始各个点的2^j辈祖先是谁 ,其中 2^j (j =0...log(该点深度))辈祖先，1辈祖先(j=0)就是父亲，2辈祖先(j=1)是父亲的父亲.....。

```
void dp()
{
//     for(int i=1;i<=n;i++)
//                 f[i][0]=fa[i];
       H=log2(n);
       for(int j=1;j<=H;j++)
                 for(int i=1;i<=n;i++)
                           f[i][j]=f[f[i][j-1]][j-1];
}
```

3.从深度大的节点上升至深度小的节点同层，如果此时两节点相同直接返回此节点，即lca。否则，利用倍增法找到最小深度的 f[a][j]!=f[b][j]，此时他们的父亲p[a][0]即lca。

```
int lca(int a,int b)
{
        if(depth[a] < depth[b])swap(a, b);
        int d = depth[a] - depth[b];
        for(int i = 0; i <= H; i++)
                if(d & (1 << i))
                        a = f[a][i];            //f[a][i]表示a的第2^i个祖先结点

        if(a==b)return a;

        for(int i=H;i>=0;i--)
                if(f[a][i]!=f[b][i])
                        a=f[a][i], b=f[b][i];

        return f[a][0];
}
```

四、离线算法——Tarjan算法

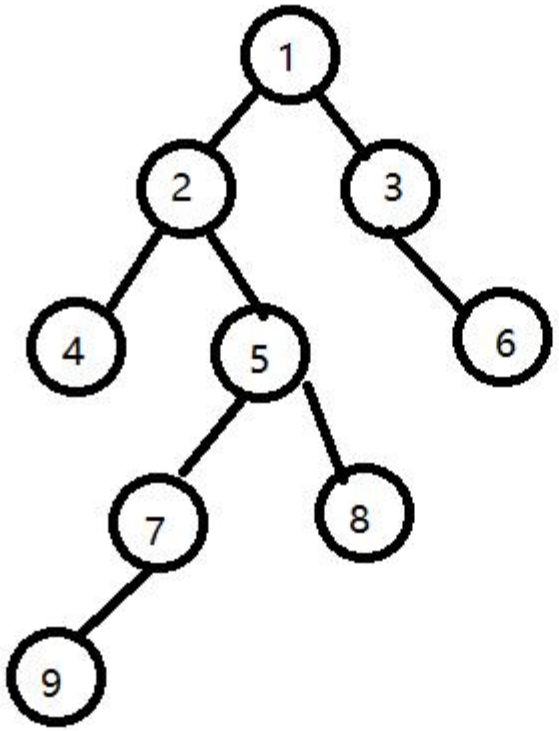
- 所谓离线算法，是指首先读入所有的询问（求一次LCA叫做一次询问），然后重新组织查询处理顺序以便得到更高效的处理方法。
- Tarjan算法是一个常见的用于解决LCA问题的离线算法，它结合了深度优先遍历和并查集，整个算法为线性处理时间。

```
Tarjan(u)//merge和find为并查集合并函数和查找函数
{
    for each(u,v)    //访问所有u子节点v
    {
        Tarjan(v);        //继续往下遍历
        merge(u,v);    //合并v到u上
        标记v被访问过;
    }
    for each(u,e)    //访问所有和u有询问关系的e
    {
        如果e被访问过，则u,e的最近公共祖先为find(e);
        //若e未被访问过则不操作
    }
}
```

- 当一个顶点的子树全部访问完并返回该顶点后，才能对涉及到该顶点的查询进行查询。

拿出纸和笔模拟一遍！

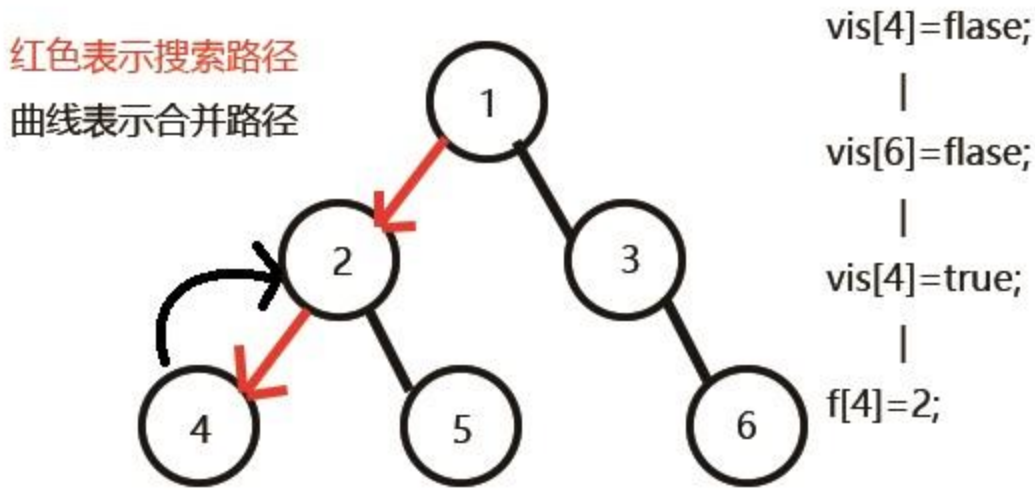
- 设我们要查找最近公共祖先的点对为（9,8）、（4,6）、（7,5）、（5,3）




```
f[1]=1;  vis[1]=flase;
f[2]=2;  vis[2]=flase;
f[3]=3;  vis[3]=flase;
f[4]=4;  vis[4]=flase;
f[5]=5;  vis[5]=flase;
f[6]=6;  vis[6]=flase;
f[7]=7;  vis[7]=flase;
f[8]=8;  vis[8]=flase;
f[9]=9;  vis[9]=flase;
```

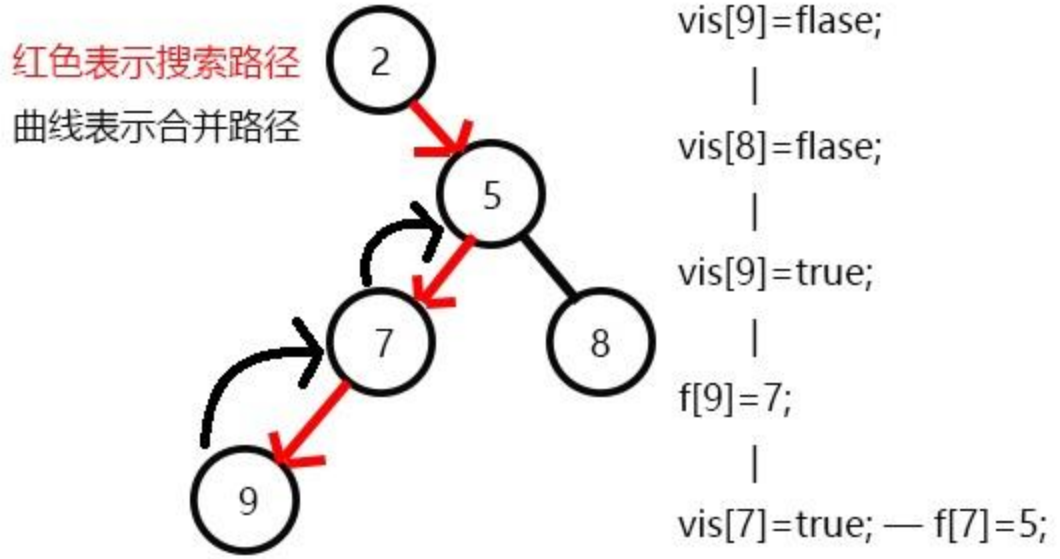
（9,8）、（4,6）、（7,5）、（5,3）

- 取1为**根节点**，**往下搜索**发现有两个儿子2和3；
- 先搜2，发现2有两个儿子4和5，先搜索4，发现4**没有子节点**，则寻找与其有关系的点；
- 发现6与4有关系，但是**vis[6]=0**，即6还没被搜过，所以**不操作**；
- 发现没有和4有询问关系的点了，返回此前一次搜索，**更新vis[4]=1**，表示4已经被搜完，**更新f[4]=2**。



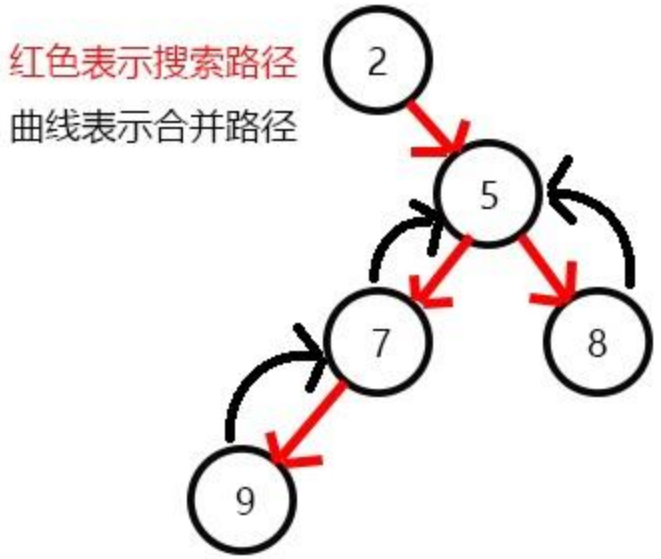
（9,8）、（4,6）、（7,5）、（5,3）

- 继续**搜5**，发现5有两个儿子7和8;
- 先**搜7**，发现7有一个子节点9，**搜索9**，发现没有子节点，寻找与其有关系的点；
- 发现8和9有关系，但是**vis[8]=0**,即8没被搜到过，所以不操作；
- 发现没有和9有询问关系的点了，返回此前一次搜索，**更新vis[9]=1**，表示9已经被搜完，**更新f[9]=7**，发现7没有没被搜过的子节点了，寻找与其有关系的点；
- 发现5和7有关系，但是**vis[5]=0**，所以**不操作**；
- 发现没有和7有关系的点了，返回此前一次搜索，**更新vis[7]=1**，表示7已经被搜完，**更新f[7]=5**



（9,8）、（4,6）、（7,5）、（5,3）

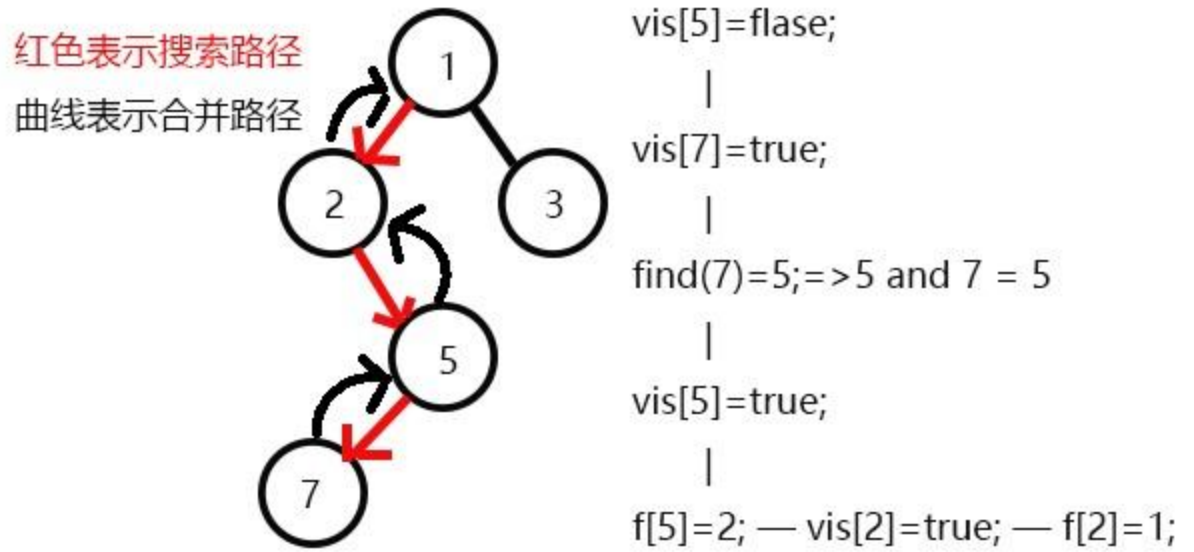
- 继续**搜8**，发现8没有子节点，则寻找与其有关系的点；
- 发现9与8有关系，**此时vis[9]=1**，则他们的**最近公共祖先为find(9)=5**；(find(9)的顺序为f[9]=7-->f[7]=5-->f[5]=5 return 5;)
- 发现没有与8有关系的点了，返回此前一次搜索，**更新vis[8]=1**；
- 表示8已经被搜完，**更新f[8]=5**，



vis[8]=flase;
|
vis[9]=true;
|
find(9)=5;=>9 and 8 = 5
|
vis[8]=true;
|
f[8]=5;

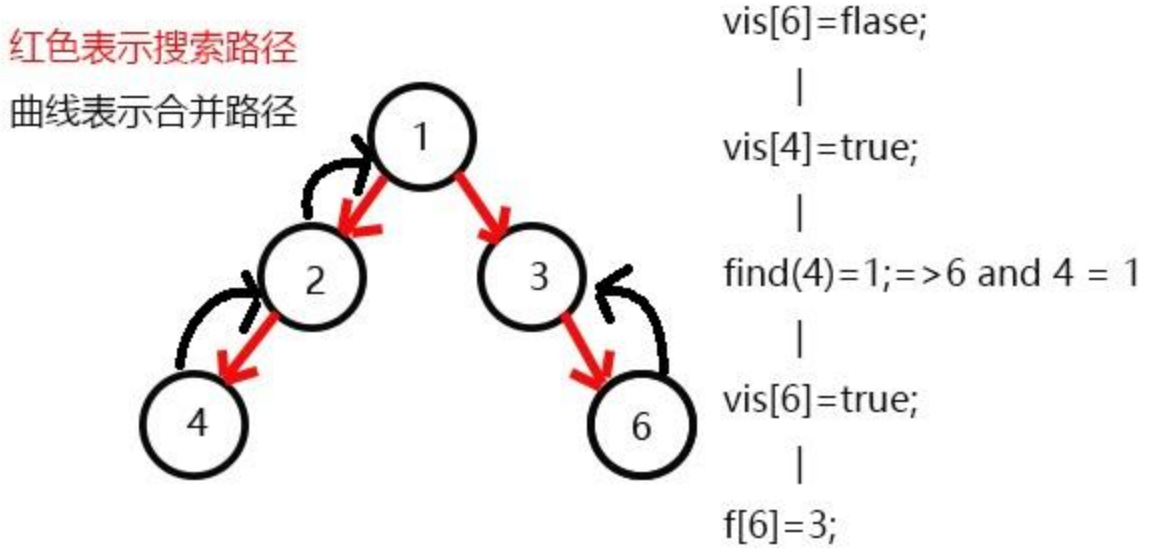
（9,8）、（4,6）、（7,5）、（5,3）

- 发现5没有没搜过的子节点了，寻找与其有关系的点；
- 发现7和5有关系，**此时vis[7]=1**，所以他们的**最近公共祖先为find(7)=5**；(find(7)的顺序为f[7]=5-->f[5]=5 return 5;)
- 又发现5和3有关系，但是**vis[3]=0**，所以**不操作**，此时5的子节点全部搜完了；返回此前一次搜索，**更新vis[5]=1**，表示5已经被搜完，**更新f[5]=2**；
- 发现2没有未被搜完的子节点，寻找与其有关系的点；
- 又发现没有和2有关系的点，则返回此前一次搜索，**更新vis[2]=1**，表示2已经被搜完，**更新f[2]=1**



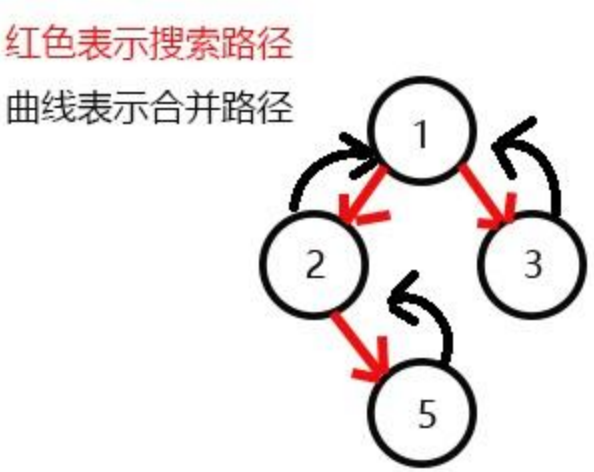
（9,8）、（4,6）、（7,5）、（5,3）

- 继续**搜3**，发现3有一个子节点6；
- **搜索6**，发现6没有子节点，则寻找与6有关系的点，发现4和6有关系；**此时vis[4]=1**，所以它们的**最近公共祖先为find(4)=1**;
- (find(4)的顺序为f[4]=2-->f[2]=1-->f[1]=1 return 1;)
- 发现没有与6有关系的点了，返回此前一次搜索，**更新vis[6]=1**，表示6已经被搜完了；**更新f[6]=3**



（9,8）、（4,6）、（7,5）、（5,3）

- 发现3没有没被搜过的子节点了，则寻找与3有关系的点；
- 发现5和3有关系，**此时vis[5]=1**，则它们的**最近公共祖先为find(5)=1**；　(find(5)的顺序为f[5]=2-->f[2]=1-->f[1]=1 return 1;)
- 发现没有和3有关系的点了，返回此前一次搜索，**更新vis[3]=1**，**更新f[3]=1**
- 发现1没有被搜过的子节点也没有有关系的点，此时可以**退出整个dfs**了。



vis[3]=flase;
|
vis[5]=true;
|
find(5)=1;=>5 and 3 =1
|
vis[3]=1;
|
f[3]=1;

参考程序

- POJ1330-3

课后习题

- HDU2586 How far away?
- POJ1470 Closest Common Ancestors
- CF E2 - Daleks' Invasion