基础算法思想 2

LeavingZ
洛谷网校

洛谷
www.luogu.com.cn

位运算

基本位运算

运算符	运算名称	含义
&	按位与	二进制两对应位均为 1 时才为 1
|	按位或	只要两个对应位中有一个 1 就为 1
^	按位异或	只有两个对应位不同时才为 1
~	取反	二进制补码中 0 和 1 全部变反
<<	二进制左移	`num << i` 表示将 `num` 的二进制左移 `i` 位
>>	二进制右移	`num >> i` 表示将 `num` 的二进制右移 `i` 位

补码：在二进制表示下，非负整数的补码为其本身，负数的补码是将其对应正数按位取反后加一。

C++ 运算符优先级：https://zh.cppreference.com/w/cpp/language/operator_precedence

位运算应用

1. 表示集合：二进制中的每一位 0/1 代表集合中某元素的存在情况。
2. 特殊题目要求的位运算

获取一个数二进制的某一位

```
int getBit(int a, int b) { return (a >> b) & 1; }
```

将一个数二进制设为 0/1 / 取反

```
int unsetBit(int a, int b) { return a & ~(1 << b); }
int setBit(int a, int b) { return a | (1 << b); }
int flapBit(int a, int b) { return a ^ (1 << b); }
```

bitset

std::bitset 是标准库中一个存储 0/1 的大小不可变的容器。

```cpp
#include <bitset>
#include <iostream>
using namespace std;
const int SIZE = 1024;
int main() {
    bitset<SIZE> set, new_set;
    //operators
    set[1] = 1;
    cout << set[1] << endl; // operator []
    cout << (set != new_set) << endl; // operator == / !=
    set |= new_set; // operator & / | / ^ / &= / |= / ^=
    // can only operate with a bitset
    set <<= 1; // operator << / >> / <<= / >>=
    // member functions
    cout << set.count() << endl; // count
    cout << set.size() << endl; // size
    cout << set.any() << set.none() << set.all(); // any none all
    set.set(1, true); set.reset(1); set.flip(1); // set reset flip
}
```

例题 – 字节交换

给定一个小于 2^{32} 的非负整数，求这个整数两个字节互换后的结果。

例题 – 高低位交换

思路

使用位运算。

```
unsigned int mask=0xff;
int x8=x<<3;
int y8=y<<3;
unsigned int nx=(n>>x8)&mask,ny=(n>>y8)&mask;
n&=(~(mask<<x8));
n&=(~(mask<<y8));
n|=(nx<<y8);
n|=(ny<<x8);
```

例题 – 集合运算

T490015

例题 – 集合运算 1

思路

使用位运算实现，这里介绍一个 `std::bitset` 做法。

使用两个 `std::bitset`，分别记录集合 A 和 B 的信息。

使用 `count()` 函数计算集合大小，使用 `&` `|` 运算符做集合的 $\cap$，$\cup$ 操作。

例题 – 集合运算 1

代码实现

```
for(int i=1;i<=n;i++)
{
    scanf("%d",&x);
    A[x]=1;
}
for(int i=1;i<=m;i++)
{
    scanf("%d",&x);
    B[x]=1;
}
int ans1=(A&B).count();
int ans2=(A|B).count();
int ans3=(A^(A&B)).count();
```

位运算性质

拆位思想

某道题目如果与位运算相关，可以尝试分开考虑每一位

And Or

一堆数字的与的结果可以考虑每一位，如果某一位上没有0，那么这一位上的结果为1，否则为0。

一堆数字的或的结果可以考虑每一位，如果某一位上没有1，那么这一位上的结果为0，否则为1。

XOR 性质

1. 偶数个相同的数字 XOR 之后结果为 0。
2. 奇数个相同的数字 XOR 之后结果为自身。

所以一个数字异或两次相同的数字等于没异或：x^y^y=x

由以上结论，拆位考虑，一堆数字异或的结果每一位上的结果如何？

考虑每一位 1 的数量的奇偶性。

XOR 性质

序列前缀 XOR 可求区间 XOR

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

$$a_L \oplus a_{L+1} \oplus a_{L+2} \cdots a_R = s_R \oplus s_{L-1}$$

利用前缀 XOR 可以解决很多序列上的异或问题

P2114

例题 P2114

拆位

依次考虑每一位，预处理出每一位初始为 0/1 的时候结果上这一位是 0 还是 1。

从高到低位贪心考虑每一位，贪心思想，尽可能让当前位为 1。

由于限制初始数字不超过 m，所以如果当前位初始为 0 不比初始为 1 差 （例如初始为 0 和初始为 1 结果相同，或者初始为 0 的时候结果为 1 并且初始为 1 的时候结果为 0）或者因为限制大小不超过 m 导致这一位初始不能为 1，我们当前位初始就取 0，否则取 1。

快速幂

快速计算 a^b

引入

```
int pow (int a, int b) {
        int x = 1;
        for (int i = 1; i <= b; ++i)
                x = 1LL * x * a % mod;
        return x;
}
```

复杂度：O(b)？

优化?

快速幂

有什么快速的方法来算 a^4 吗？

快速幂

有什么快速的方法来算 a^4 吗？

$$a^4 = (a^2)^2$$

平方 再平方

快速幂

有什么快速的方法来算 a^4 吗？

$$a^4 = (a^2)^2$$

平方 再平方

如果继续平方的话会得到 $a^1, a^2, a^4, a^8, a^{16}$ ...

也就是可以得到 a^{2^k}

快速幂

继续平方的话会得到 $a^1, a^2, a^4, a^8, a^{16}$ ...

可以得到 a^{2^k}

于是我们可以对 a^b 的指数转为二进制

假设 $b = (1001101)_2 = 2^0 + 2^2 + 2^3 + 2^6$，容易发现我们应该把每个二进制为 1 的位置乘起来。

例如 $(1001101)_2$ 就相当于 $a^{2^0} \cdot a^{2^2} \cdot a^{2^3} \cdot a^{2^6}$

快速幂

继续平方的话会得到 $a^1, a^2, a^4, a^8, a^{16}$ ...

可以得到 a^{2^k}

于是我们可以对 a^b 的指数转为二进制

假设 $b = (1001101)_2$，容易发现我们应该把每个二进制为 1 的位置乘起来，例如 $(101101)_2$ 就相当于 $a^{2^0} \cdot a^{2^2} \cdot a^{2^3} \cdot a^{2^6}$

容易发现最多 $\log_2 b$ 项，且每一项都是形如 a^{2^k} 的形式，容易得到。

代码实现

```
int qpow (int a, int b) {
        int x = 1;
        while (b) {
                if (b & 1)
                        x = a * x % p;
                a = a * a % p;
                b >>= 1;
        }
        return x;
}
```

非递归快速幂

例题-P1226 【模板】快速幂

给三个整数 a, b, p ， 求 $a^b \ mod \ p$

$0 \leq a, b < 2^{31}, a + b > 0, 2 \leq p < 2^{31}$

分治

分而治之

什么是分治

分治（英语：Divide and Conquer），字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

什么是分治

分治（英语：Divide and Conquer），字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

分治步骤

分治的步骤：

1.分解原问题为结构相同的子问题。
2.进行递归求解，分解到某个容易求解的边界之后，直接得到答案。
3.将子问题的解合并成原问题的解。

例题-归并排序

我们可以利用分治思想来实现一种 $O(n \log_2 n)$ 的排序——归并排序。

例题-归并排序

我们可以利用分治思想来实现一种 $O(n \log_2 n)$ 的排序——归并排序。

原问题——对范围 $[l, r]$ 进行排序

例题-归并排序

我们可以利用分治思想来实现一种 $O(n \log_2 n)$ 的排序——归并排序。

原问题——对范围 $[l, r]$ 进行排序

子问题——对范围 $[l, mid], [mid + 1, r]$ 分别排序

例题-归并排序

我们可以利用分治思想来实现一种 $O(n \log_2 n)$ 的排序——归并排序。

原问题——对范围 $[l, r]$ 进行排序

子问题——对范围 $[l, mid], [mid + 1, r]$ 分别排序

边　界——范围内只有一个数字，不用排序

例题-归并排序

我们可以利用分治思想来实现一种 $O(n \log_2 n)$ 的排序——归并排序。

原问题——对范围 $[l, r]$ 进行排序

子问题——对范围 $[l, mid], [mid + 1, r]$ 分别排序

边　界——范围内只有一个数字，不用排序

合　并——将分别有序的范围 $[l, mid], [mid + 1, r]$ 合并为有序的范围 $[l, r]$

例题-归并排序

如何合并两个有序序列为一个新的有序序列？

每次取两个有序序列的最小值中更小的那个，放入新序列末尾。

例题-P1908 逆序对

- 给一个序列 a，定义每一对满足 $i < j \ and \ a_i > a_j$ 的数对唯一对逆序对，求序列 a 中逆序对个数

- $N \leq 5e5$

例题-P1908 逆序对

- 分治?
- 把一个序列分成两半，分别计算左右半边内部的的逆序对数目，再计算 i 在左半，j 在右半的逆序对数目

例题-P1908 逆序对

- 分治?
- 把一个序列分成两半，分别计算两个序列的逆序对树木，再再计算 i 在左半，j 在右半的逆序对数目

- 怎么计算跨越区间的部分?
- 归并排序

倍增

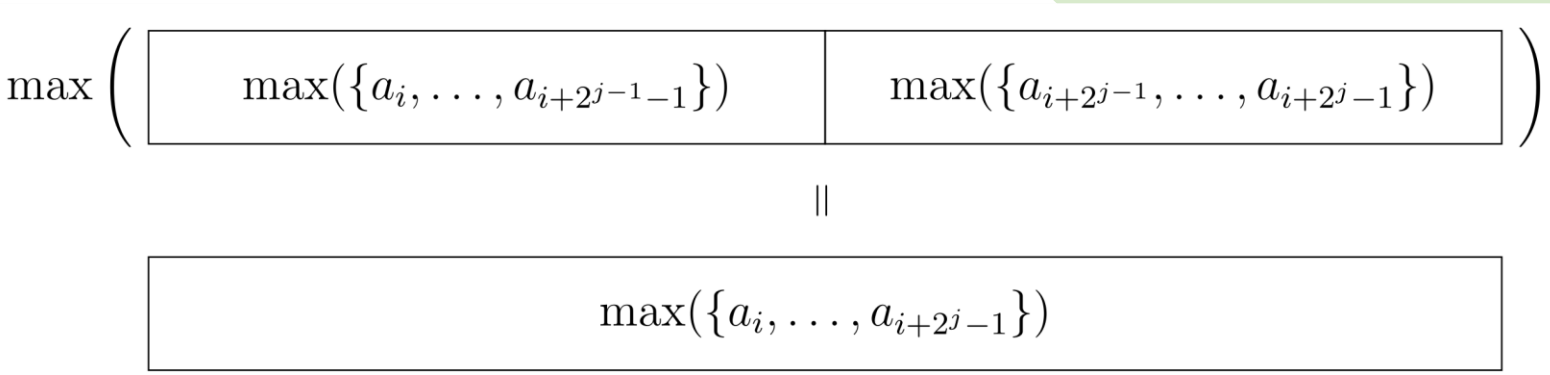
倍增

- 预处理倍增思想：从 $2^0 = 1$ 开始进行预处理，每次将两个 2^{j-1} 的部分合并成为 2^j 的部分，从而处理出所有的 2 的幂次的信息。
- 询问时，将任意的范围进行二进制拆分，合并至多 $\log_2 S$个信息组成答案。

ST表

- $f[i][j]$，指的是在序列的第 i 项，**向后 2^j 个元素所包含序列间的最大值**。
- 预处理转移：

$$f[i,j] = \max\{f[i,j-1], f[i+2^{j-1}, j-1]\}$$



ST表

- 查询，对于区间 l~r:

$$\text{长度}s = \log_2(r - l + 1)$$

$$ans = \max\{f[l, s], f[r - 2^s + 1, s]\}$$

$$\max = \max(\max_l, \max_r) = 14$$

$$\max_r = f(4, 2) = 13$$

$$\max_l = f(2, 2) = 14$$

0	13	14	4	13	1	5	7

P2866

倍增/ST表

P2866

```
for(int i=1;i<=n;i++)
{
    int pos=i+1;//已经确认了 [i,pos-1] 没有挡住 i
    for(int k=J;k>=0;k--)//倒序循环
//如果 2^k 跳过去还能看得到表示答案肯定比 2^k 大，那么第 k 位一定为 1、
//如果顺序循环，2^k 跳过去没被挡住答案肯定比 2^k 大没错，但是这时候第 k 位不一定为 1
    {
        if(pos+(1<<k)-1<=n)
//[pos,pos+2^k-1]没有挡住表示答案肯定比 2^k 大，那么第 k 位一定为 1
        {
            if(st[k][pos]<h[i])
//当前确认[i,pos-1] 没有挡住
//现在 [pos,pos+2^k-1] 没有挡住，那么 [i,pos+2^k-1] 没有挡住
//[i,pos-1]->[i,pos+2^k-1] ---- pos+=2^k
                pos+=(1<<k);
        }
    }
//[i,pos-1] 不包括 i 有 pos-1-i 个
    c[i]=pos-i-1;
    ans+=c[i];
}
```

树上倍增

类似的，我们可以在树上处理出关于祖先的信息。

例如，我们可以预处理出 $fa[u][i]$ 表示节点 u 的第 2^i 级祖先是谁。

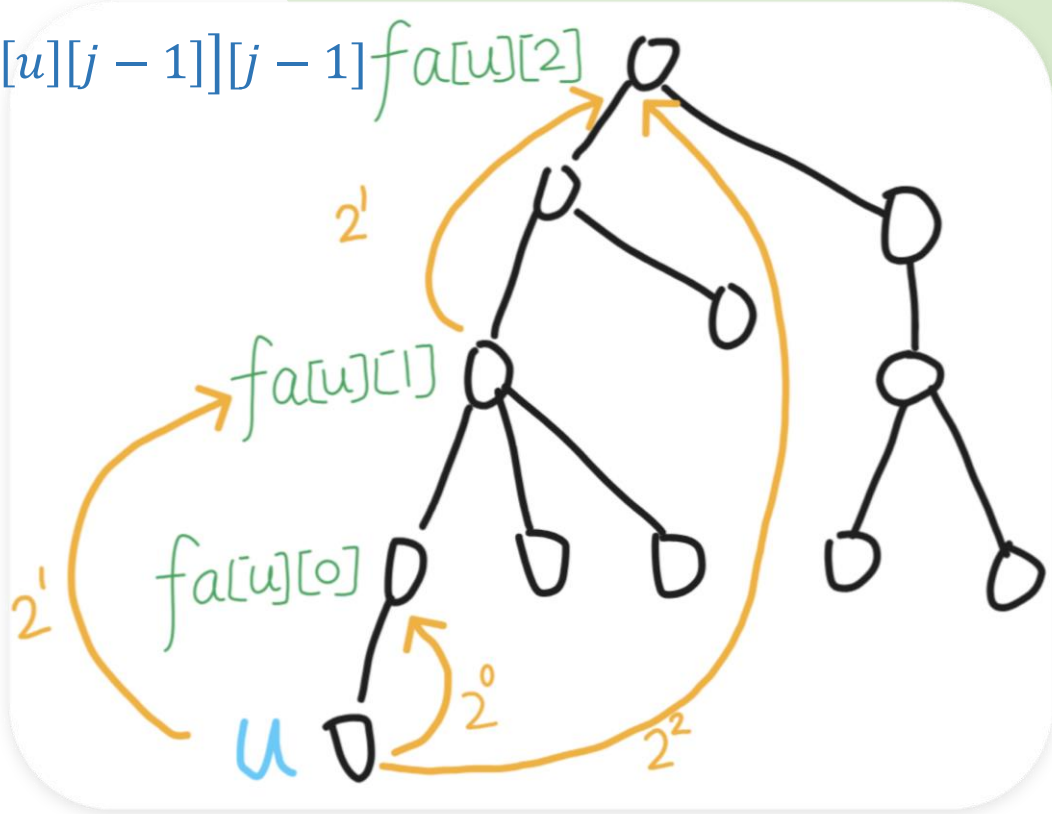
树上倍增

类似的，我们可以在树上处理出关于祖先的信息。

例如，我们可以预处理出 $fa[u][i]$ 表示节点 u 的第 2^i 级祖先是谁。

初始时，$fa[u][0]$ 即为节点 u 的父亲。

对于 $j > 0$ 有 $fa[u][j] = fa[fa[u][j-1]][j-1]$



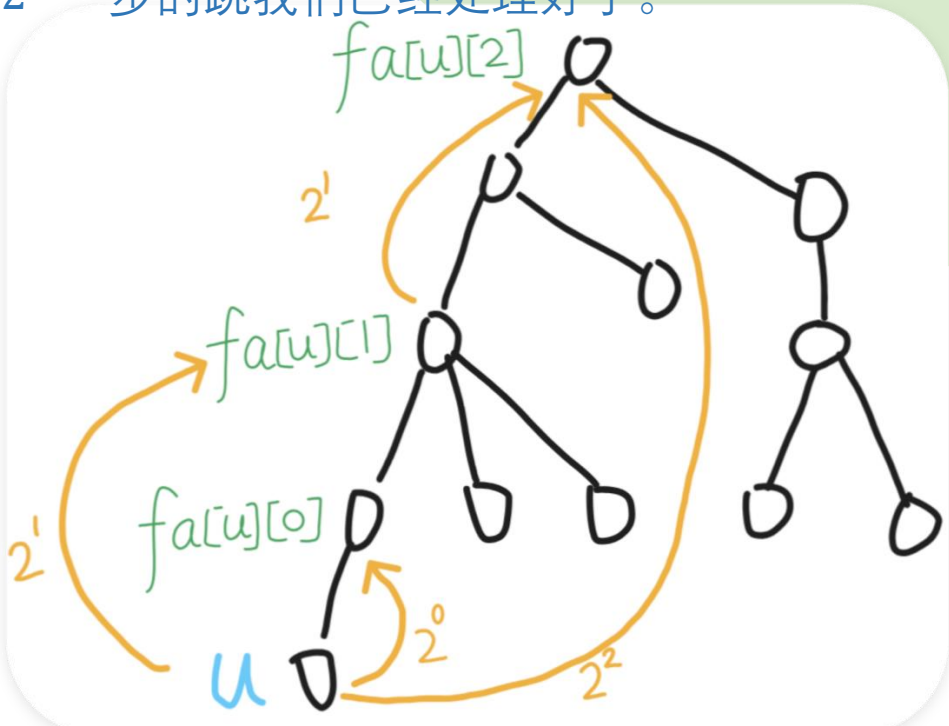
树上倍增

$fa[u][i]$ 表示节点 u 的第 2^i 级祖先是谁。

初始时，$fa[u][0]$ 即为节点 u 的父亲。

对于 $j > 0$ 有 $fa[u][j] = fa[fa[u][j-1]][j-1]$

要求节点 u 的 2^j 级祖先，即节点 u 上跳 2^j 步之后所在的节点，可以把这一跳拆分成两个 2^{j-1} 步的跳，而 2^{j-1} 步的跳我们已经处理好了。



树上倍增

要求节点 u 的 2^j 级祖先，即节点 u 上跳 2^j 步之后所在的节点，可以把这一跳拆分成两个 2^{j-1} 步的跳，而 2^{j-1} 步的跳我们已经处理好了。

要求节点 u 的 k 级祖先，就把 k 进行二进制拆分，拆分成若干个 2 的幂次之和（看二进制下为 1 的位），例如 $23 = 2^4 + 2^2 + 2^1 + 2^0 = (10111)_2$，然后分别跳 $2^4, 2^2, 2^1, 2^0$ 步，容易发现，拆分出来的幂次至多有 $\log_2 k$ 项，因此我们可以在 $O(\log_2 k)$ 的时间内求出 k 级祖先。

树上倍增

类似的，我们可以在树上处理出关于祖先的信息。

例如，我们可以预处理出 $fa[u][i]$ 表示节点 u 的第 2^i 级祖先是谁。

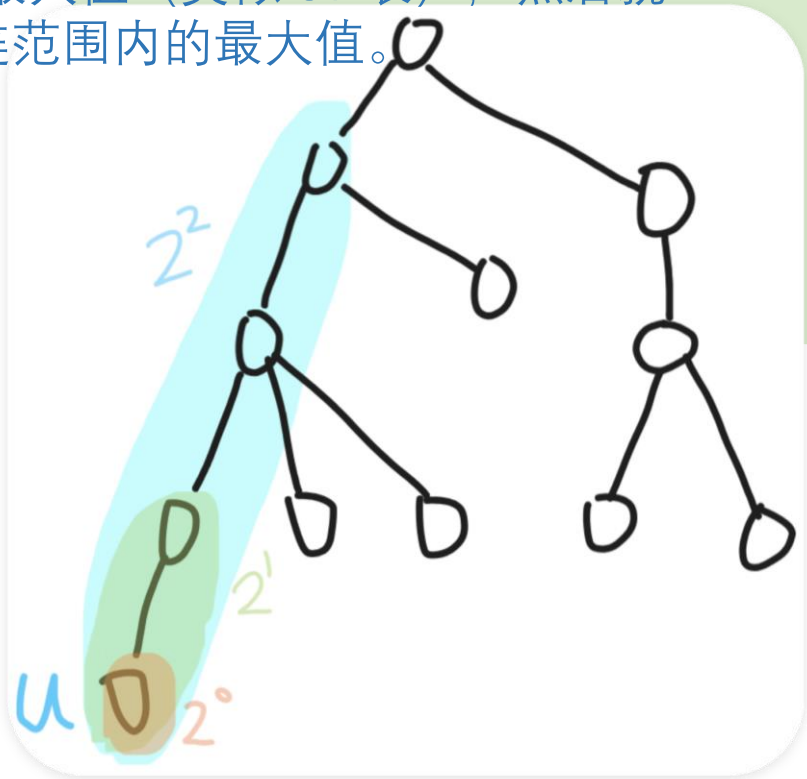
例题：求树上 k 级祖先

树上倍增

类似的，我们可以在树上处理出关于祖先的信息。

例如，我们可以预处理出 $fa[u][i]$ 表示节点 u 的第 2^i 级祖先是谁。

不失一般性的，我们可以推广这种处理信息，例如还可以求出 $mx[u][i]$ 表示节点 u 以及其向上 2^i-1 级祖先内的最大值（类似 ST 表），然后就可以求出任意点到其任何一个祖先这一条链范围内的最大值。

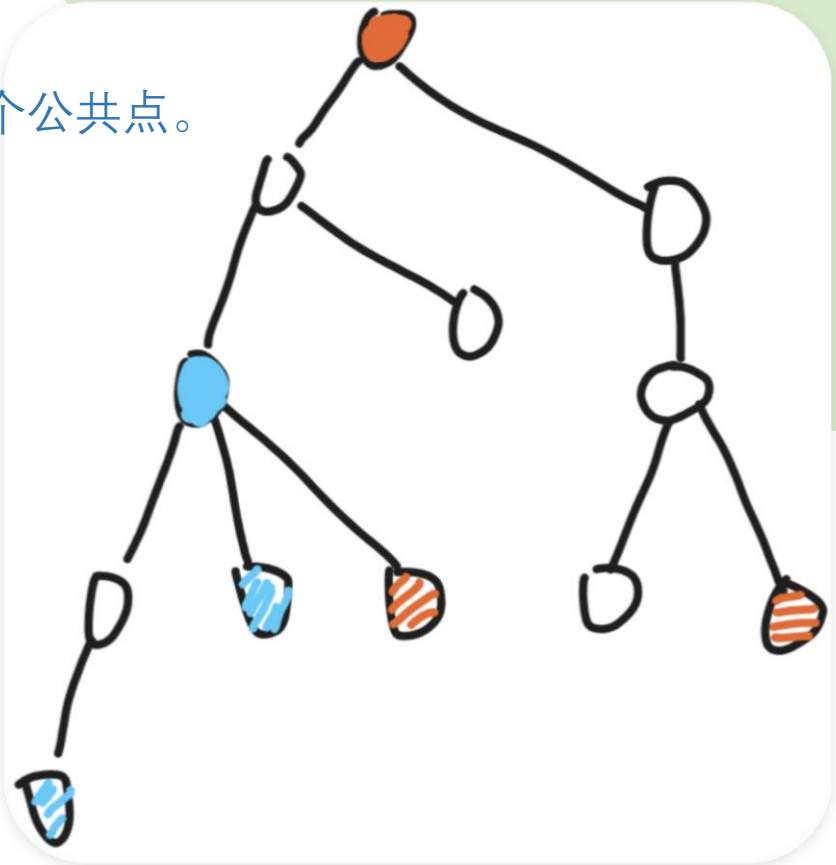


LCA

最近公共祖先简称 LCA（Lowest Common Ancestor）。两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个。

也可以理解成，两个点的最近公共祖先 LCA，就是从一个点走到另一个点的路径上拐弯的那个点。

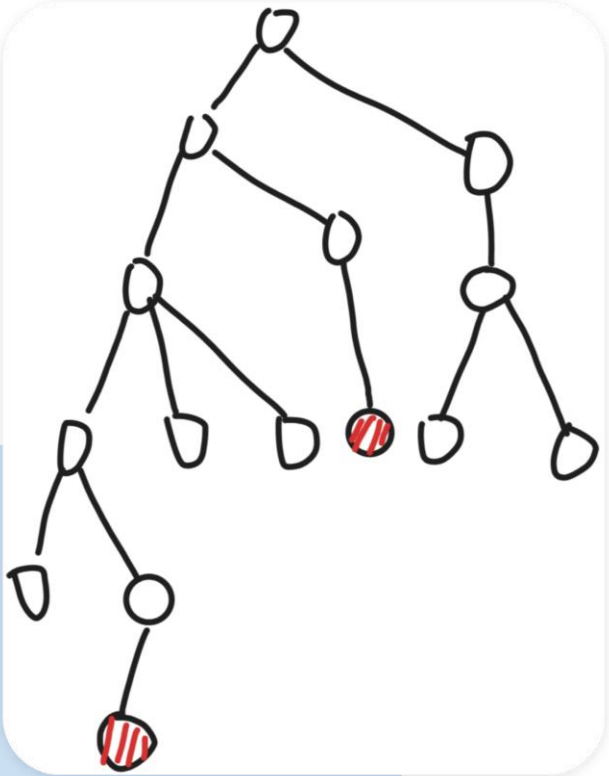
或者说是两个点向树根走的路径上的第一个公共点。

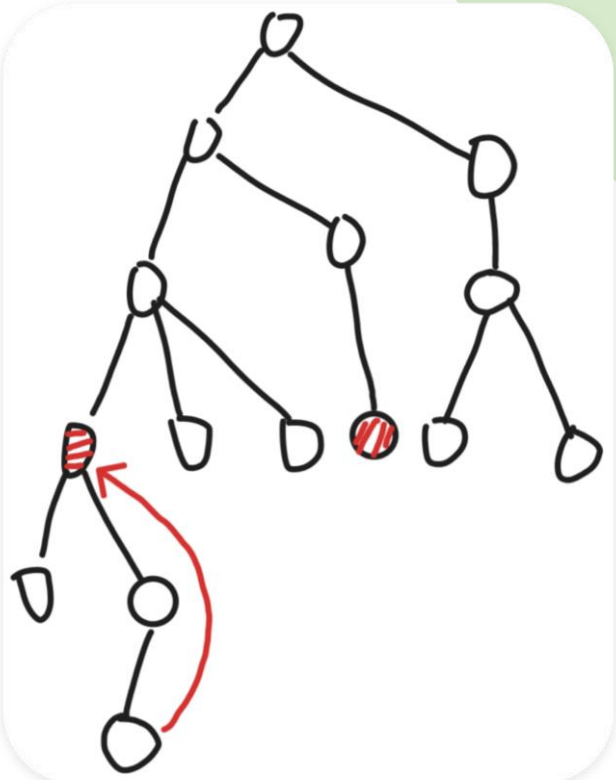


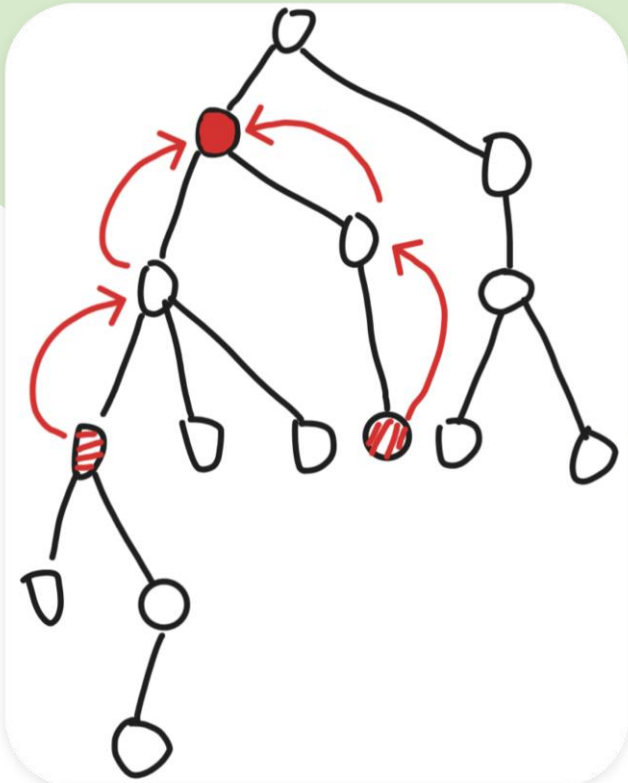
LCA

朴素的 LCA 求法

先提到同一个深度，然后暴力一步步上跳直到相遇。







LCA

可以利用倍增加速上跳过程

LCA

```
int LCA(int x,int y)
{
    if(dep[x]<dep[y]) swap(x,y);
    int d=dep[x]-dep[y];//深度差
    for(int k=0;k<=J;k++)
    if(d&(1<<k)) x=fa[k][x];//提到相同深度
    if(x==y) return x;//重合直接返回
    //求最晚不相遇的点，求出来之后再走一步就相遇了
    for(int k=J;k>=0;k--)//尝试往上跳 2^k 步
    if(fa[k][x]!=fa[k][y])
    //没重合表示与最后不相遇点的深度差大于等于 2^k，那么第 k 位一定为 1，一定要往上跳一次 2^k
    //如果重合了表示深度差小于 2^k，不跳
    {
        x=fa[k][x];
        y=fa[k][y];
    }
    //最后求出的是最后不相遇的点，还得往上走一步才是LCA
    return fa[0][x];
}
```

LCA

学会求 LCA 以后，我们就可以对任意的路径求出最大值了

$u \to v$
$u \to lca(u,v) \to v$
$u \to lca(u,v) + v \to lca(u,v)$