Rbook 代码模板书

Small But Powerful

by Rainboy @ 2020-06-28

1 排序算法 4
1.1 冒泡排序 4
1.2 快速排序 5
1.3 归并排序 5
2 搜索算法 6
2.1 序 6
2.2 深度优先：DFS 6
2.2.1 深度优先搜索 6
2.2.2 剪枝 7
2.3 广度优先：BFS 8
2.3.1 01BFS 8
2.3.2 A 星算法 9
2.4 IDA* 10
2.4.1 IDA* 10
3 字符串算法 11
3.1 字符串 hash 11
3.2 kmp 算法 12
3.2.1 KMP 算法 12
3.3 字典树 / Trie 12
3.3.1 字典树 12
3.4 AC 自动机 13
4 分治算法 14
4.1 二分查找 14
4.1.1 二分查找 15
5 动态规划 15
5.1 线型 DP 15
5.1.1 LIS 最长不下降子序列 15
5.1.1.1 LIS 最长不下降子序列 15
5.1.2 LCS 最长公共子序列 18
5.1.2.1 LCS 最长公共子序列 18
5.2 背包 DP 20
5.2.1 01 背包 20

5.2.2 完全背包 20
5.2.3 多重背包 20
5.2.4 分组背包 21
6 树相关算法 21
6.1 LCA 最近公共祖先 21
6.1.1 tarjan 求 lca - 离线 21
6.1.2 LCA 在线算法 - 树上倍增 22
6.2 最小生成树 23
6.2.1 kruskal 23
6.2.2 次小生成树 24
6.3 树的重心与直径 25
6.3.1 树的重心 26
6.3.2 树的直径 27
6.4 DFS 序 27
6.4.1 DFS 序：基础 27
6.5 树链剖分 28
6.5.1 lca 29
6.6 树上启发合并 33
6.6.1 dsu on tree 入门 33
7 图论算法 35
7.1 最短路问题 35
7.1.1 Floyed-WarShall 算法 35
7.1.1.1 floyed 35
7.1.1.2 floyed 最小环 35
7.1.2 Dijkstra 36
7.1.3 Bellman-ford 36
7.1.4 SPFA 37
7.1.5 spfa_dfs 37
7.1.6 k 短路径 38
7.2 强连通分量 39
7.2.1 Tarjan 39
7.3 无向图连通性 41

7.3.1 图的割点 41
7.3.2 图的割边 41
7.3.3 双联通分量 42
7.3.3.1 点连通分量 42
7.3.3.2 边连通分量 44
7.3.4 割点，割边，点双，边双四合一模板 46
7.4 拓扑排序 50
7.4.1 拓扑排序 50
7.5 欧拉图与哈密顿图 54
7.5.1 欧拉回路 54
8 数据结构 55
8.1 RMQ/ST/ 区间最值 55
8.1.1 RMQ 区间最值 55
8.2 树状数组 55
8.2.1 树状数组：基础 55
8.2.2 逆序对 57
8.2.3 区间修改，单点查询 58
8.2.4 区间增减 区间查询 58
8.2.5 树状数组 区间最值 59
8.2.6 二维树状数组 60
8.2.7 综合模板 60
8.3 并查集 62
8.3.1 并查集 62
8.4 线段树 62
8.4.1 单点更新 62
8.4.2 成段更新 64
8.4.3 线段树合并 66
8.5 🔒 「动态开点线段树」 与 「权值线段树」 「感谢」 67
8.5.1 动态开点线段树 67
8.6 🚧 🔒 主席树 68
8.6.1 主席树：入门 68

8.7 平衡树 71
8.7.1 替罪羊树 [Scapegoat Tree] 71
8.7.2 Splay 入门 73
8.7.3 Treap 78
8.7.4 fhq-treap 81
8.8 link cut tree(LCT) 82
8.8.1 「LCT 入门」序 82
9 其它算法 83
9.1 快读 83
9.2 序列中和不超过 K 的对数 84
9.3 离散化 85

1 排序算法

1.1 冒泡排序

```
void bubble_sort(int a[],int n){
    int i,j;
    int tmp;
    for(i=1;i<=n-1;i++) //n个数,要进行n-1趟排序
        for(j=1;j<=n-i;j++){//第i趟排序的最后一个下标:n-i
            if(a[j] > a[j+1]){
                tmp =a[j];
                a[j] =a[j+1];
                a[j+1]=tmp;
            }
        }
}
```

```
    //冒泡排序
    for (i=1;i<=n-1;i++){ //n-1轮排序
        for(j=1;j<=n-i;j++){
            if( a[j] > a[j+1]){
                cnt++;
                swap(a[j],a[j+1]);
            }
        }
    }
```

1.2 快速排序

```
void quicksort(int l, int r) {
    int i, j, t, tmp;
    if(l > r) return;
    tmp = a[l]; //tmp中存的就是基准数
    i = l, j = r;
    while(i != j) { //顺序很重要，要先从右边开始找
        while(a[j] >= tmp && i < j) j--;
        //再找右边的
        while(a[i] <= tmp && i < j) i++;
        //交换两个数在数组中的位置
        if(i < j) swap(a[i],a[j]);
    }
    //最终将基准数归位
    a[l] = a[i];
    a[i] = tmp;
    quicksort(l, i-1);//继续处理左边的，这里是一个递归的过程
    quicksort(i+1, r);//继续处理右边的 ，这里是一个递归的过程
}
```

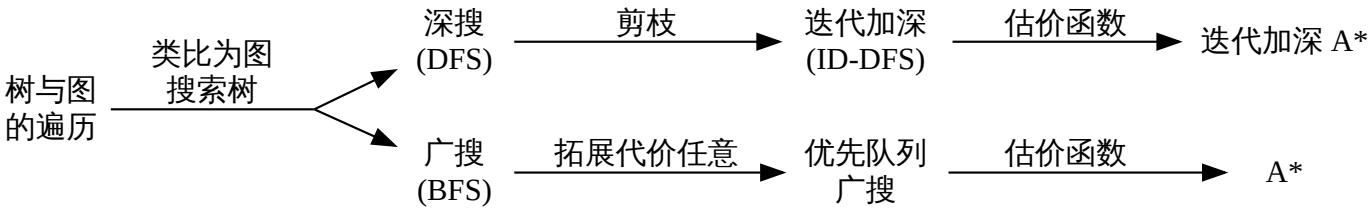
1.3 归并排序

```
void merge_sort(int s,int t){
        //s =start t=T
    int mid,i,j,k;

    if(s==t) return ; //如果区间只有一个数,就返回

    mid = (s+t)>>1; //取中间的点
    merge_sort(s,mid);
    merge_sort(mid+1,t);

    i=s,k=s,j=mid+1;

    while(i<=mid && j<=t){
        if( a[i] <=a[j]){
            tmp[k]=a[i];k++;i++;
        } else {
            tmp[k]=a[j];j++;k++;
        }
    }

```

```
    while(i<=mid) { tmp[k]=a[i];k++;i++;}
    while(j<=t)   { tmp[k]=a[j];k++;j++;}

    for(i=s;i<=t;i++)
        a[i]=tmp[i];
}
```

2 搜索算法

2.1 序



2.2 深度优先：DFS

2.2.1 深度优先搜索

```
void dfs( state arguments){

    if(到达边界条件)
        结束函数

    for(i=0;i<=max;i++){

        state_next = opt
        //

        dfs(state_next)

    }
}
```

2.2.2 剪枝

剪枝: 通过某些判断，搜索树的某些**枝条**可以不用遍历，那我们可以把它剪掉 (不搜索), 故称为剪枝.

剪枝的原则

- 正确性
- 准确性
- 高效性

剪枝分类

优化搜索顺序

如图 1, 如果要求深度最低的叶子结点，明显先向左边走，先能达到最值点 2.

排除等效冗余

在搜索的过程中，如果我们能够判定从搜索树的当前解都会沿着几条不同的分支到达的子树是等效的，那么只要对其中的一条分支进行搜索.

可行性剪枝

当搜索到某一个点 (状态) 时，及时对当前点进行检查，如果发现分支已经不可能达到递归边界了 (得到答案), 就回溯.

发走路的过程中，发现当前路径不可能达到终点，那就回溯.

特别的当某个点被限制在一个区间内的时候，此时可行性剪枝被称为**上下界剪枝**

相关题目:

- luogu P1025 数的划分

最优性剪枝

在搜索的过程中，如果当前花费的代价已经超过**已经得到的解中的最优解**, 无论如何如当前点出发**都不可能更新答案了**! 执行回溯.

如图 1, 如果要求深度最低的叶子结点，明显先向左边走，先能达到最值点 2, 随后只要超过点 2 深度的深度，就回溯.

- A* BFS + 估价函数
- IDA 迭代加深
- IDA* 迭代加深 + 估价函数

记忆化

已经搜索过的状态，记录下来，下一次遇到这个状态是，直接返回结果。是 DP 的搜索写法.

一个简单的题目:luogu P1028 数的计算

2.3 广度优先：BFS

2.3.1 01BFS

使用双端队列 $deque$

- 从0 边拓展到的点放入队首
- 从1 边拓展到的点放入队尾
- 每次从队首取点，直到队列为空

```
void _01bfs(int state,int sx,int sy){
    /* 清空标记 */
    memset(vis,0,sizeof(vis));
    typedef struct { int x,y,step; } node;
    deque<node> q;
    q.push_back({sx,sy,0}); //加入起点

    while( !q.empty()){
        node h = q.front();
        q.pop_front();
        if( vis[h.x][h.y])
            continue;
        vis[h.x][h.y] = 1; //标记

        dis[state][h.x][h.y] = h.step;

        int i;
        for(i = 0; i < 4;i++ ){
            int nx = h.x  + fx[i][0];
            int ny = h.y  + fx[i][1];
            if( in_map(nx,ny) && !vis[nx][ny] && _map[x][y] != '#' ){
                if( _map[x][y] == '0')
                    q.push_front({nx,ny,h.step});
                else
                    q.push_back({nx,ny,h.step+1})
            }
        }
    }
}

```

2.3.2 A 星算法

- 把起点添加 $open\_list$
- 当$open_list$ 不空
  - 找到 open_list 中 F 值最小的点 u
  - if u 是终点
    - 返回：路径已经找到
  - 移动点 u 到 $close\_list$
  - 对 u 相邻的所有点 v
    - 如果 v **不可通过**或者已经在 $close\_list$ 中，略过它。反之如下。
    - 如果 v 不在 $open\_list$ 中
      - 记录 v 的 (F, G, H) 值
      - 把 v 加入 $open\_list$
      - $father[v] = u$
    - 如果 v 在 $open\_list$ 中，重新计算 F 值，如果 F 值更小:
      - 更新 v 的 (F, G, H) 值
      - $father[v] = u$
- $open\_list$ 已经空，没有找到目标格，路径不存在。

```
//F = g + h 估价函数 g实际 h估计
struct node {
    int g,h;
    friend bool operator<(const node &a,const node &b){
        return a.g+a.h > b.g+b.h;
    }
};
bool astart(){
    node start;
    priority_queue<node> q;
    q.push(start);

    while( !q.empty() ){
        node head = q.top(); q.pop();
        if( head is 终点) return 1;
        if( head in vis ) continue;
        vis[head] = 1;

        for( each v in <head,v> ){
            if( v in vis) continue;
            node t = get_f(v); 计算v在F值
            q.push(t); //放入优先队列
```

```
        }

    }
    return -1; //没有找到目标
}

```

2.4 IDA*

2.4.1 IDA*

- A* 用在 BFS 上，A* = 优先队列 + 估价函数
- IDA *用在 DFS 上，IDA* = 迭代加深 + 估价函数

```
bool iddfs(int d,int maxd, args...){
    if( d == maxd+1){
        if( 符合题意 ) return true;
    }

    for(int abble = 1;  abble <= 最大次数; abble++ ){
        if( 估价函数(d,maxd) 符合条件  )
            iddfs(d+1,maxd, args...)
    }
}

//迭代加深
for( int maxd = 1 ;  ; maxd++){
    if(iddfs(dep=1,maxd, ... ) ) break;
}
```

3 字符串算法

3.1 字符串 hash

```
const int maxn = 1e5+5;
namespace HASH {
    typedef unsigned long long ULL;
    const int seed = 133331;
    const char start = 'A';
    ULL b[maxn],sum[maxn];
    void init_b(){
        b[0] = 1;
```

```
        for(int i=1;i<=maxn;i++) b[i] = b[i-1]*seed;
    }
    void get_sum(char s[],int len){
        sum[0] = 0;
        for(int i=1;i<=len;i++) sum[i] = sum[i-1]*seed + (ULL)(s[i]-start+1);
    }

    ULL get_hash(char s[],int len){
        ULL s = 0;
        s = s*seed + (ULL)(s[i]-start+1);
    }

    ULL get_hash_range_from_sum(int l,int r){
        return sum[r] - sum[l-1]*b[r-l+1];
    }
}

```

3.2 kmp 算法

3.2.1 KMP 算法

```
const int kmp_max_len = 1e5;
struct KMP{
    int la,lb;
    char a[kmp_max_len],b[kmp_max_len];
    int next[kmp_max_len];
    int cnt =0; //匹配字符串的数量

    void deal_next(){
        int i,j=0;
        next[1] = 0;
        for(i=2;i<=lb;++i){
            while(j && b[j+1] != b[i])  j = next[j];
            if( b[j+1] == b[i]) j++;
            next[i] = j;
        }
    }
    void kmp(){
        int i,j=0;
        for(i=1;i<=la;++i){
            while(j && b[j+1] != a[i]) j = next[j];
            if( b[j+1] == a[i]) j++;
```

```
            if( j == lb){
                //printf("%d\n",i-lb+1);
                cnt++;
                j=next[j];
            }
        }
    }
};
```

3.3 字典树 / Trie

3.3.1 字典树

```
struct trie {
  int nex[100000][26], cnt;
  bool exist[100000];  // 该结点结尾的字符串是否存在

  void insert(char *s, int l) {  // 插入字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) nex[p][c] = ++cnt;  // 如果没有，就添加结点
      p = nex[p][c];
    }
    exist[p] = 1;
  }
  bool find(char *s, int l) {  // 查找字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) return 0;
      p = nex[p][c];
    }
    return exist[p];
  }
};
```

3.4 AC 自动机

```
namespace AC {

    //trie树,每个单词出现的次数,失配指针
```

```
4     int ch[maxn][26],cntword[maxn],next[maxn],tot=1;
5     int bo[maxn]; // 是否是单词
6
7     void ac_init(){
8         tot=1;  //结点从1开始编号
9         memset(bo,0,sizeof(bo));
10        memset(ch,0,sizeof(ch));
11        memset(next,0,sizeof(next));
12    }
13
14    void build(char *s){  // 建立ch树
15        int len = strlen(s), u = 1;
16        for(int i=0; i<len; ++i){
17            int c = s[i] -'a';
18            if( !ch[u][c]) ch[u][c] = ++tot;
19            u = ch[u][c];
20        }
21       bo[u]++;
22    }
23    void  bfs_next(){
24        for(int i =0;i<=25;++i) ch[0][i] = 1;
25        queue<int> q; q.push(1); //队列
26        next[1] = 0;              //根的失配指针
27        while( !q.empty()){
28            int u = q.front(); q.pop();
29            for(int i = 0;i <= 25; ++i){
30                if( !ch[u][i]) ch[u][i] = ch[next[u]][i]; // 优化
31                else {
32                    q.push(ch[u][i]);
33                    int v = next[u];
34                    next[ ch[u][i] ] = ch[v][i];
35                }
36            }
37        }
38    }
39
40    void find( char *s){
41        int u = 1, len=strlen(s);
42        for(int i = 0; i <=len ;++i){
43            int c = s[i] - 'a';
44            int k = ch[u][c];
45            while( k > 1){
46                ans += bo[k];
47                bo[k] = 0;
```

```
                k = next[k];
            }
            u = ch[u][c];
        }
    }
}

```

4 分治算法

4.1 二分查找

4.1.1 二分查找

```
#include <cstdio>

int a[] = {1,2,3,4,5};

int bsearch(int l,int r,int key){

    while (l<=r){
        int m = (l+r)>>1;
        if(a[m] == key)
            return m;
        if( key < a[m])
            r = m-1;
        else
            l = m+1;
    }
    return -1;
}

int main(){
    int len_a = sizeof(a)/sizeof(a[0]);

    int ret = bsearch(0,len_a-1,2);
    printf("%d\n",ret);
    return 0;
}
```

5 动态规划

5.1 线型 DP

5.1.1 LIS 最长不下降子序列

5.1.1.1 LIS 最长不下降子序列

```
#include <cstdio>

int a[1000];
int n;
int f[1000];

int main(){
    scanf("%d",&n);
    int i;
    int j;
    for (i=1;i<=n;i++){
        scanf("%d",&a[i]);
    }

    //每个点的f值不可能小于1
    for(i=1;i<=n;i++) f[i] =1;

    int max = 1;//这里是1,想想为什么
    for (i=2;i<=n;i++){
        for(j=1;j<i;j++){
            if( a[j] <= a[i] && f[i] < f[j]+1){
                f[i] = f[j]+1;
                if(max < f[i])
                    max = f[i];
            }
        }
    }

    printf("%d",max);

    return 0;
}
```

```
a: 7 2 9 3 4 10 6 1  原数组
f: 1 1 2 2 3  4 4 1  f[i],以a[i]为结尾的LIS的值
```

c [i]: 长度为 i 的最长不下降子序列的最小右端值数组	C[1]	C[2]	C[3]	C[4]	C[6]

c [i]: 长度为 i 的最长不下降子序列的最小右端值数组	C[1]	C[2]	C[3]	C[4]	C[6]
处理到前 0 个数	∞	∞	∞	∞	∞
处理到前 1 个数	7	∞	∞	∞	∞
处理到前 2 个数	2	∞	∞	∞	∞
处理到前 3 个数	2	9	∞	∞	∞
处理到前 4 个数	2	3	∞	∞	∞
处理到前 5 个数	2	3	4	∞	∞
处理到前 6 个数	2	3	4	10	∞
处理到前 7 个数	2	3	4	6	∞
处理到前 8 个数	1	3	4	6	∞

数据：

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

```
// 求最长不下降子序列
#include <cstdio>
#include <cstring>

int n;
int a[100];
int c[100],f[100];

//[l,r)
int upper_bound(int l,int r,int key){
    while(l < r){
        int m  = (l+r)>>1;
        if( c[m] <= key)
            l = m+1;
        else
            r = m;
    }
    return l;
}

void init(){
    memset(c,0x7f,sizeof(c));
```

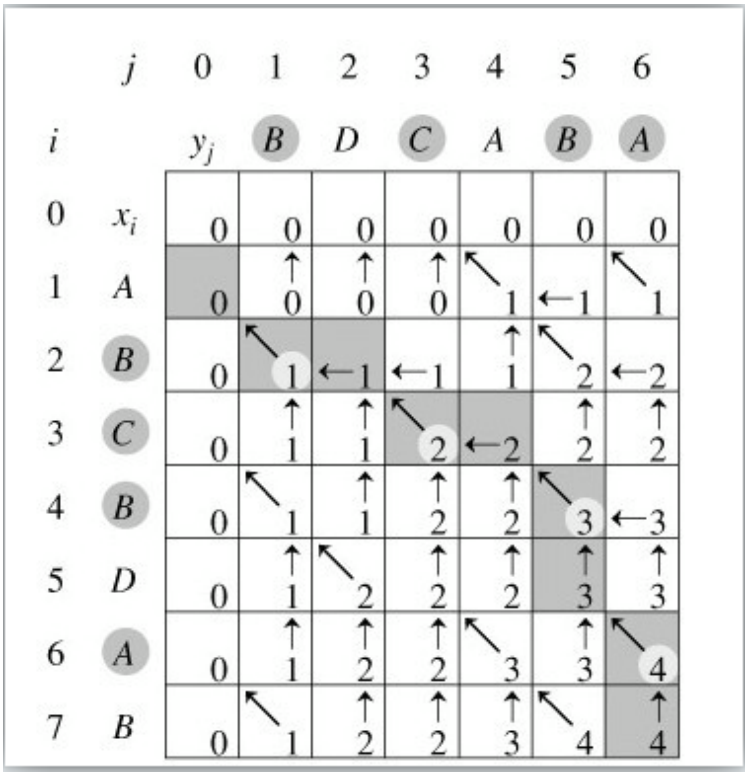
```
    scanf("%d",&n);
    int i;
    for (i=1;i<=n;i++){
        scanf("%d",&a[i]);
    }
}

void lis(){
    int i,j,k;
    c[1] = a[1];
    f[1] = 1;
    for(i=2;i<=n;i++){
        int pos = upper_bound(1,i,a[i]);
        pos--;
        f[i] = pos+1;
        if( c[ f[i] ]  > a[i])
             c[ f[i] ]  = a[i];
    }
}
int main(){
    init();
    lis();
    int i;
    for (i=1;i<=n;i++){
        printf("%d %d\n",i,f[i]);
    }

    return 0;
}
```

5.1.2 LCS 最长公共子序列

5.1.2.1 LCS 最长公共子序列

```
#include <cstdio>
#include <cstring>

char s1[5010];
char s2[5010];
int l1,l2;

int f[5010][5010] = {0};

```

```
int max(int a,int b){
    if( a>  b)
        return a;
    return b;
}

int lcs(){
    int i,j;

    for(i=1;i<=l1;i++)
        for(j=1;j<=l2;j++){
            if( s1[i] == s2[j]){
                f[i][j] = f[i-1][j-1]+1;
            }
            else
                f[i][j] = max(f[i][j-1],f[i-1][j]);
        }
    return f[l1][l2];
}
int main(){
    scanf("%s",s1+1);
    scanf("%s",&s2[1]);
    l1 = strlen(s1+1);//求长度
    l2 = strlen(s2+1);

    int m = lcs();
    printf("%d",m);

    return 0;
}
```



LCS 转 LIS

使用，求下面的两个序列的 LCS

```
2 1 3
1 2 3
```

因为第二个序列是有序且上升的，第一个序列和第二个序列的 LCS 序列必然也是上升的，也就是求第一个序列的 LIS.

所以:**LCS 转 LIS ：将序列 A 和 B 当中的相同字母配对都找出来，呈现成索引值数对，再以特殊规则排序，最后找 LIS ，就是 A 和 B 的 LCS **

5.2 背包 DP

5.2.1 01 背包

```
for i=1->N //前i个物品
    for j=C->w[i] //容量从大到小
        f[j] = max(f[j],f[j-w[i]]+v[i])
```

5.2.2 完全背包

```
for i = 1->N          //枚举前i个物品
    for v=c[i]->V     //枚举背包容量
```

```
        f[v] = max{f[v],f[v-c[i]]+w[i]}
```

5.2.3 多重背包

```
            for (int k=1; k<=c; k<<=1) { //<<右移 相当于乘二
                value[count] = k*v;
                size[count++] = k*s;
                c -= k;
            }
            if (c > 0) {
                value[count] = c*v;
                size[count++] = c*s;
            }
```

5.2.4 分组背包

```
int i,j,k;
for(i=1;i<=N;i++)
    for(K=C;k>=0 ;k--){   //先枚举容量
        for(j=1;j<=num[i];j++){ //再枚举组内物品
            if(W[i][j] > k) continue;
            f[k] = max{
                f[k],
                f[k-W[i][j]] + V[i][j]
            }
        }
    }
```

核心思想: 按组 -> 容量 -> 组内物品的顺序枚举，这样就保证了：当前容量，当前组内物品 j

- 当前格子是组内前 j-1 个物品得到的价值，
- 前面的格子是前 i 组物品得到的价值

6 树相关算法

6.1 LCA 最近公共祖先

6.1.1 tarjan 求 lca - 离线

```
namespace Trajan_lca {

```

```
    int qhead[maxn],q_cnt = 0;
    struct Query { int u,v,next,id; }query[maxe];
    void query_queue_init(){   q_cnt = 0; memset(qhead,-1,sizeof(qhead)); }
    void addQuery(int u,int v){ query[q_cnt]={u,v,head[u],q_cnt};
qhead[u]=q_cnt++; }

    int fa[maxn],vis[maxn];
    void find(int u){ //路径压缩
        if( fa[u] == u) return u;
        return fa[u] = find(fa[u]);
    }

    void tarjan_lca(int u){
        fa[u] = u;
        for(int i = head[u]; ~i ;i = e[i].next){
            int v = e[i].v;
            trajan_lca(v);
            fa[v] = u;
        }
        vis[u] = 1; //标记 放这里的原因就是 求lca(6,6)这种点
        for(int i = qhead[u] ;~i ; i =query[i].next){
            int v = query[i].v;
            if( vis[v]) ans[query[i].id] = find(v);
        }
    }
}

```

6.1.2 LCA 在线算法 - 树上倍增

```
//============= 树上倍增 BEG
namespace BZ_LCA {
    const int SIZ = 35;
    int f[maxn][SIZ+1],len[maxn][SIZ+1],dep[maxn];
    using namespace xlx1;
    void dfs_init(int u ,int d,int fa,int w){
        dep[u] = d; f[u][0] = fa; len[u][0] = w;
        for(int i= 1;i<=SIZ;++i){
            f[u][i] = f[f[u][i-1]][i-1];
            len[u][i] = len[u][i-1]+len[f[u][i-1]][i-1];
        }
        for(int i=head[u];~i;i=e[i].next){
```

```
            int v =e[i].v ,w = e[i].w;
            if( v == fa) continue;
            dfs_init(v,d+1,u,w);
        }
    }
    int find_lca(int u,int v){
        int sum = 0;
        if( dep[u] < dep[v]) swap(u,v); //保证u点的深度深
        for(int i=SIZ;i>=0;--i){
            if( dep[f[u][i]] < dep[v]) continue; //不跳的区域
            sum+=len[u][i];
            u = f[u][i];
        }
        if(u == v) return sum;
        for(int i=SIZ;i>=0;--i){
            if( f[u][i] == f[v][i] ) continue;
            sum+=len[u][i];
            sum+=len[v][i];
            u = f[u][i];
            v = f[v][i];
        }
        return sum+len[u][0]+len[v][0];
    }
}
//============= 树上倍增 END

```

6.2 最小生成树

6.2.1 kruskal

1. 不同的最小生成树中，每种权值的边出现的个数是确定的
2. 不同的生成树中，某一种权值的边连接完成后，形成的联通块状态是一样的

https://blog.csdn.net/clover_hxy/article/details/69397184

```
// 向量星 略

/* ================ 并查集 ================*/
const int maxn = 1e5+5;
int fa[maxn];
//并查集 初始化
inline void bcj_init(int x){ for(int i=1;i<=x;i++) fa[i] = i; }

```

```
//查找 and 路径压缩
int find(int x){
    if( x == fa[x]) return x;
    return fa[x] = find(fa[x]);
}
/* ================= 并查集 end =================*/

bool cmp( edge a,edge b) {return a.w < b.w;}
// 传入点数，返回mst的值，不连通返回-1
int Kruskal(int n){
    bcj_init(n); //初始化
    sort(e+1,e+1+edge_cnt,cmp); //对边从小到大排序
    int ans = 0,cnt = 0; //答案，选边的数量

    for(int i = 1; i<=edge_cnt ; ++i){
        int u =e[i].u, v =e[i].v, w =e[i].w;
        int f1 =find(u),f2 = find(v);
        if( f1 != f2){ //不再同一个集合
            ans+=w;
            cnt++;
            fa[f2] = f1;
        }
        if( cnt == n-1) break;
    }
    if( cnt < n -1) return -1;
    return ans;
}
```

6.2.2 次小生成树

首先求出 MST, **枚举每一条不在 MST 上的边，然后把这条边放到 MST 上，这样一定会形成环，删除这个环上的最大边 (非新加入的那条边), 就会形成一个新的生成树**，最这些新的生成树中值最小的，就是**次最小生成树**

1. 先求出mst 的值
2. 合并并查集的时候: **求出** $maxd$
3. 枚举非mst 上的边$< u, v >$, 加入到mst 上，得到小的生成树的值:

$$new\_value = mst\_value + w(u, v) - maxd(u, v)$$

```
/* 边集数组 */
struct _e{
    long long u,v,w,vis; //vis 是否是mst上的边
```

```
}e[maxm<<1];

//G[i] 表示集合i上的点
vector<int> G[maxn];

long long second_mst(){
    long long mst,second_mst=0x7fffffffffffffff;
    int i,j,k=0;
    /* 初始化 并查集 */
    for (i=1;i<=n;i++){
        fa_bcj[i] = i;
        G[i].push_back(i);
    }

    for(i=1;i<=m;i++){
        int f1 = find(e[i].u);
        int f2 = find(e[i].v);
        if( f1 != f2){
            mst += e[i].w;
            fa_bcj[f1] = f2;
            e[i].vis = 1; // 这条边在 MST 上

            /* 求maxd*/
            long long &u = e[i].u,&v = e[i].v,&w = e[i].w;

            for( auto x1 : G[f1]){
                for( auto x2 : G[f2]){
                    maxd[x1][x2] = maxd[x2][x1] = w;
                }
            }

            /* 把集和 f1 合并到f2 上,不再关心f1集合*/
            G[f2].insert( G[f2].end(),G[f1].begin(),G[f1].end());

            k++;
            if( k == n-1) break; //选n-1条边
        }
    }

    for(i=1;i<=m;i++){
        if( !e[i].vis){
            long long t = mst + e[i].w - maxd[e[i].u][e[i].v];
            second_mst = min(second_mst,t);
        }
```

```
    }
    return second_mst;
}

```

6.3 树的重心与直径

6.3.1 树的重心

定义 1：

> 树若以某点为根，使得该树最大子树的结点数最小，那么这个点则为该树的重心，一棵树可能有多个重心。

定义 2：

> 以这个点为根，那么所有的子树（不算整个树自身）的大小都不超过整个树大小的一半

上面两个定义是等价的

树的重心的性质：

- i. 树上所有的点到树的重心的距离之和是最短的，如果有多个重心，那么总距离相等。
- ii. 插入或删除一个点，树的重心的位置最多移动一个单位 (一条边的距离)。
- iii. 若添加一条边连接 2 棵树，那么新树的重心一定在原来两棵树的重心的路径上。
- iv. 一棵树最多有两个重心，且相邻。

```
const int maxn=20100;
int n,father;
int siz[maxn];//siz保存每个节点的子树大小。
bool vist[maxn];
int CenterOfGravity=0x3f3f3f3f,minsum=-1;//minsum表示切掉重心后最大连通块的大小。
vector<int>G[maxn];
void DFS(int u,int x){//遍历到节点x, x的父亲是u。
    siz[x]=1;
    bool flag=true;
    for(int i=0;i<G[x].size();i++){
        int v=G[x][i];
        if(!vist[v]){
            vist[v]=true;
            DFS(x,v);//访问子节点。
            siz[x]+=siz[v];//回溯计算本节点的siz
            if(siz[v]>n/2) flag=false;//判断节点x是不是重心。
```

```
        }
    }
    if(n-siz[x]>n/2) flag=false;//判断节点x是不是重心。
    if(flag && x<CenterOfGravity) CenterOfGravity=x,father=u;//这里写
x<CenterOfGravity是因为本题中要求节点编号最小的重心。
}
```

6.3.2 树的直径

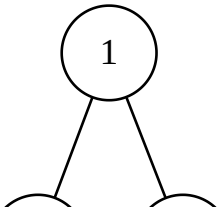
定义: 什么是树的直径: **树上面的最长的一条路径**, 被称为**树的直径**. 也叫树的最长路径，最远点对.

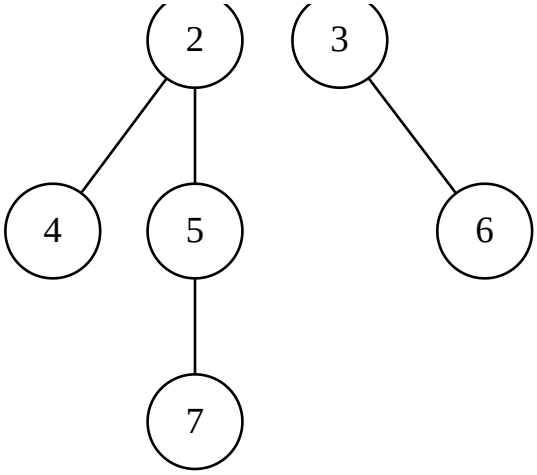
两次 dfs 的模板

```
//TLP:tree_longest_path
namespace TLP {
    using namespace xlx1;
    typedef long long ll;
    ll node,dis;
    //len: 根到点u的距离
    void dfs(int u,int fa,int len){
        if(dis <len ) node = u, dis = len;
        for(int i = head[u];i!=-1;i=e[i].next){
            int v = e[i].v;
            if( v == fa) continue;
            dfs(v,u,len+e[i].w);
        }
    }
    void work(){
        dis=-1;dfs(1,0,0);   //第1次dfs
        dis=-1;dfs(node,0,0);//第2次dfs
        //得到最长的直径dis
    }
}
```

6.4 DFS 序

6.4.1 DFS 序：基础





所谓的 dfs 序，就是按 dfs 的访问的顺序形成的序列.

如上图可以形成，序列 1:

1	2	4	5	7	3	6

```
int start[maxn],end[maxn];
int dep[maxn],fa[maxn];
int dfs_clock = 0;
int dfs(int d,int u,int pre){
    dep[u] = d;
    fa[u] = pre;
    start[u] = ++dfs_clock;
    int i;
    for(i = head[u]; ~i ; i = e[i].next){
        int v= e[i].v;
        if( v != pre){
            dfs(d+1,v,u);
        }
    }
    end[u] = dfs_clock;
}
```

```
const int maxn = 1e5+5;
int in[maxn],out[maxn];
inline bool isChild(int a,int b)
    if( start[b]<= start[a] && end[a] <= end[b])
        return 1;
    return 0;
}
```

6.5 树链剖分

6.5.1 lca

```cpp
#include <cstdio>
#include <cstring>

#define maxn 100

int m,n;
int root;

int head[maxn];
struct edge {
    int next;
    int v;
}E[maxn<<1]; //存两遍边
int cnt = 0;

void addedge(int x,int y){
    cnt++;
    E[cnt].v = y;
    E[cnt].next = head[x];
    head[x] = cnt;
}

int son[maxn];
int top[maxn];
int dep[maxn];
int fa[maxn];
int size[maxn];

void dfs1(int u,int pre,int d){
    dep[u] = d;
    fa[u] = pre;
    size[u] = 1;
    int i;
    for(i=head[u];i!=-1;i=E[i].next){
        int v = E[i].v;
        dfs1(v,u,d+1);
        size[u] += size[v];
        if(son[u] == -1 || size[v] > size[ son[u] ])
            son[u] = v;
    }
```

```
41 }
42
43 void dfs2(int u,int sf){
44     top[u] = sf;
45     if(son[u] != -1)
46         dfs2(son[u],sf);
47     else
48         return ;
49     int i;
50
51     for(i=head[u];i!=-1;i=E[i].next){
52         int v = E[i].v;
53         if( v!= son[u] && v != fa[u])
54             dfs2(v,v);
55     }
56 }
57
58 void swap(int &x,int &y){
59     int t = x;
60     x = y;
61     y =t;
62 }
63
64 //找到lca(x,y)
65 int find(int x,int y){
66     //找到两个点的重链的顶端点
67     int f1 = top[x],f2 = top[y];
68     int tmp = 0;
69     while(f1 != f2){
70
71         //从深度较深的点 向上爬
72         if( dep[f1] < dep[f2]){
73             swap(f1,f2);
74             swap(x,y);
75         }
76         //交换后 y所在重链的 dep[ top[y] ] < dep[ top[x] ]
77         //x top[x] 较深
78
79         //跨链
80         x =  fa[f1];
81         f1 = top[x];
82     }
83
84     //返回较浅的那个点
```

```
    if( dep[x] > dep[y])
        swap(x,y);
    return x;
}

int main(){
    memset(head,-1,sizeof(head));
    memset(son,-1,sizeof(son));
    scanf("%d%d",&m,&root);
    int i;
    for (i=1;i<=m;i++){
        int x,y;
        scanf("%d%d",&x,&y);
        addedge(x,y);
    }
    dfs1(root,root,1);
    dfs2(root,root);

    scanf("%d",&n);
    int x,y;
    for(i=1;i<=n;i++){
        scanf("%d%d",&x,&y);
        int ans = find(x,y);
        printf("%d\n",ans);
    }

    return 0;
}
```

练习题目 2

题目地址：【模板】最近公共祖先（LCA）

代码

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define maxn 500005
using namespace std;
int m,n,root;
struct edge{
```

```
    int next;
    int v;
}E[maxn<<1]; //存两遍边
int head[maxn];
int son[maxn],top[maxn],size[maxn],fa[maxn],dep[maxn];
int cnt=0;
void addEdge(int x,int y){
    cnt++;
    E[cnt].v=y;
    E[cnt].next=head[x];
    head[x]=cnt;
}
void dfs1(int u,int pre,int d){
    dep[u]=d;
    fa[u]=pre;
    size[u]=1;
    for(int i=head[u];i!=-1;i=E[i].next){
        int v=E[i].v;
        if( v == pre ) continue;
        dfs1(v,u,d+1);
        size[u]+=size[v];
        if(son[u]==-1 || size[v]>size[son[u]]){
            son[u]=v;
        }
    }
}
void dfs2(int u,int sf){
    top[u]=sf;
    if(son[u]!=-1){
        dfs2(son[u],top[u]);
    }
    else return ;
    for(int i=head[u];i!=-1;i=E[i].next){
        int v=E[i].v;
        if(v!=son[u] && v != fa[u])
            dfs2(v,v);
    }
}
int find(int x,int y){
    int f1=top[x],f2=top[y];
    while(f1!=f2){
        if(dep[f1]<dep[f2]){
            swap(f1,f2);
            swap(x,y);
```

```
52          }
53          x=fa[f1];
54          f1=top[x];
55      }
56      if(dep[x]>dep[y]){
57          swap(x,y);
58      }
59      return x;
60  }
61  int main(){
62      memset(son,-1,sizeof(son));
63      memset(head,-1,sizeof(head));
64      scanf("%d%d%d",&n,&m,&root);
65      for(int i=1;i<=n-1;i++){
66          int a,b;
67          scanf("%d%d",&a,&b);
68          addEdge(a,b);
69          addEdge(b,a);
70      }
71      dfs1(root,root,1);
72      dfs2(root,root);
73      for(int i=1;i<=m;i++){
74          int a,b;
75          scanf("%d%d",&a,&b);
76          printf("%d\n",find(a,b));
77      }
78      return 0;
79  }
```

6.6 树上启发合并

6.6.1 dsu on tree 入门

树上启发式合并 $dsu\ on\ tree$ 跟 dsu (并查集) 是没啥关系，它是用来解决一类树上询问问题，一般这种问题有两个特征

1. 只有对子树的询问
2. 没有修改

对子树进行暴力 $O(n^2)$

但是，划分轻重

dfs 进入结点 u 时:

- dfs 所有的轻儿子为根的子树，单是不会保留轻儿子对**信息记录**的贡献
- dfs 重儿子为根的子树，保留重儿子对**信息记录**的贡献
- 此时:
  i. 暴力统计 u 及其 u 的轻为根的子树上的所有点，统计信息，加入**信息记录**
  ii. 根新 u 结点为根的子树的答案
- 回溯退出 u, 此时 - 如果 u 为重儿子，什么也不做 (保留了 u 为根的子树对**信息记录**的贡献) - 如果 u 为轻儿子，删除 u 为根的子树上的所有点对**信息记录**的贡献

主体框架:

```
//暴力统计u结点,和轻儿子的信息
void dfs_count(int u){
    dfs_count(v not hson[u])
}

//删除u结点为根的子树对 信息记录 的贡献
void del_count(int u){
}

//u: 当前结点,fa: 父结点,keep: 是否保留u的贡献
void dsu_on_tree(int u,int fa,bool keep){
    // 1. 递归算轻儿子树
    for(v in u.lighson){
        dsu_on_tree(v,u,false);
    }

    // 2. 递归算重儿子树
    if( son[u] ){
        dsu_on_tree(son[u], u, true);
        flag_hson = son[u];
    }
    // 3. 暴力统计u,和u的轻儿子子树的贡献
    dfs_count(u,fa);
    ans[u] = max_color_sum; //得到u结点子树的答案
    if( !keep ){
        // 从u回溯,删除子树上的所有点的贡献
        //! 会把记录信息的数组清零
        del_count(u, fa);
        // 清空 必须放这里,想一想,只有重儿子的情况
        max_color_sum = max_color_cnt = 0;
    }
}
```

7 图论算法

7.1 最短路问题

7.1.1 Floyed-WarShall 算法

7.1.1.1 floyed

一句话算法:
floyd 算法的原理: i,j 经过中间点 k 的最短路径，不停的枚举 k

```
void floyd(){
    int k,i,j;
    for(k=1;k<=n;k++){ //k在最外层,枚举中间点
        for (i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                if(f[i][k]+f[k][j]<f[i][j]){ //松弛法,如果能更小,那就更小
                    f[i][j] = f[i][k]+f[k][j];
                }
    }
}
```

7.1.1.2 floyed 最小环

```
for(k=1;k<=n;k++) {
    for(i=1;i<k;i++) // 此时[i,j]之间的最短路还不经过k
        for(j=i+1;j<k;j++) // 为什么是i+1,不用枚举f[i][j]后又枚举f[j][i],对称性
            if(f[i][j]+m[i][k]+m[k][j]<ans)
                ans=f[i][j]+m[i][k]+m[k][j];
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(f[i][k]+f[k][j]<f[i][j])
                f[i][j]=f[i][k]+f[k][j];
}
```

7.1.2 Dijkstra

```
priority_queue <pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > >q;
void dijkstra(int s){
    for(int i=1;i<=n;i++){ dis[i]=inf; }
    dis[s]=0;
    q.push(make_pair(0,s));
```

```
    while(!q.empty()){
        int now=q.top().second;
        q.pop();
        if(vis[now])continue;
        vis[now]=1;
        for(int i=head[now];i!=-1;i=e[i].next){
            if(dis[e[i].v]>dis[now]+e[i].w){
                dis[e[i].v]=dis[now]+e[i].w;
                q.push(make_pair(dis[e[i].v],e[i].v));
            }
        }
    }
}
```

7.1.3 Bellman-ford

有 n 个点，每一个用一个点用更新周围的点，最多更新 n-1 次，就得到了每个点的 dis 值

```
void bellman_ford(){
    int i,j;
    for(i=1;i<=n-1;i++)//进行n-1轮操作
        for(j=1;j<=m;j++){
            int &ss = u[j],&tt=v[j],&ww = w[j];//引用

            if(dis[ss] > dis[tt] + ww)
                dis[ss] = dis[tt] +ww;

            /* 如果是无向图 要反过求一次 */
            if(dis[tt] > dis[ss] + ww)
                dis[tt] = dis[ss] +ww;
        }
    //代码完成
}
```

7.1.4 SPFA

- SPFA 是对 bellman-ford 算法的队列优化
- 一个点被更新了，它还有可能更新周围的点，入队

```
void spfa(){
    push(s);
    dis[s] = 0;
    inQueue[s] = 1;
```

```
    pre[s] = -1;


    int i;
    while( empty() == false){
        int t = pop();  //取队首
        inQueue[t] = 0; // 不在队中

        for(i = first[t];i !=-1;i = edge[i].next){
            int tv = edge[i].v;
            int tw = edge[i].w;
            if( dis[tv] > dis[t] + tw){ //更新
                dis[tv] = dis[t] + tw;

                /* 不在队列中,就加入队列 */
                if( inQueue[tv] == 0){
                    push(tv);
                    inQueue[tv] = 1;
                }
            }
        }
    }
}
```

7.1.5 spfa_dfs

所以利用这个性质 + dfs, 如果存在正环，则一个点可以被访问多次，也就是说，当点 u 还在栈中时候，还能再次入栈.

伪代码：

```
/*  求负环为什么dis清0？为什dis,ins只要清一次？
 * */
namespace spfa_dfs{
    using namespace xlx1;
    bool ins[maxn]; //在栈内
    int dis[maxn]; // double ,long long 根据题意更改
    bool dfs(int u){ //找负环
        ins[u] = 1;
        for( int i = head[u]; ~i; i = e[i].next){
             int v = e[i].v,w=e[i].w;
             if( dis[v] > dis[u]+w){
                 dis[v] = dis[u]+w;
```

```
                if( ins[v] || dfs(v) ) return true;
            }
        }
        ins[u] = 0;
        return 0;
    }
    bool wk(){
        memset(dis,0,sizeof(dis));
        memset(ins,0,sizeof(ins)); // bool的全局变量可能不全是0
        for(int i=1;i<=n;++i){
            if( dfs(i) ) return 1;
        }
        return 0;
    }
}
```

7.1.6 k 短路径

```
const int maxn = 1e5+5;
int dis[maxn];
struct node_for_astar_k {
    int v,w;
    friend bool operator<(const node_for_astar_k &a,const node_for_astar_k &b){
        return a.w+dis[a.v] >  b.w+dis[b.v];
    }
};

int astart_Kshort_path(int s,int t){

    /* 起点和终点相同是 */
    if( s == t){
        k++;
    }
    q.push(qnode(s,0));
    while(!q.empty()){

        qnode h = q.top();
        q.pop();

        int now = h.v;
        if( now ==t){
            cnt++;
            if( cnt == k ) return h.w;
```

```
        }

        int i;
        for(i = head[now]; i!= -1;i = e[i].next){
            int &v = e[i].v;
            int &w = e[i].w;
            q.push(qnode(v,w+h.w));
        }

    }

    return -1;
}

```

7.2 强连通分量

7.2.1 Tarjan

- 当 (u,v) 是树枝边时$low[u] = min(low[u], low[v])$
- 当 (u,v) 是回边，且另一个点没有被输出 (在 stack 内) 时，$low[u] = min(low[u], dfn[v])$
- 当 dfs 退出点 u, 判断 u 是不是强连通分量的根，$dfn[u] == low[u]$

算法实现过程:

- 对于 dfs 中的每个点
- 初始化 `dfn[x]` 和 `low[x]`
- 对 `x` 的所有临接点'v':
- 如果没有被访问过，则访问 v, 同时维护 low [x]
- 如果被访问过，但没有被输出，就维护 low [x]
- 如果 `dfn[x] == low[x]`, 输出

模板:

```
int color[maxn],color_cnt = 0;    // 每个点的颜色,也就是属于的连通分量的编号
bool instack[maxn];
stack<int> sta;     // 栈
void tarjan(int u) {
    dfn[u]=low[u]=++dfn;        // 为节点u设定次序编号和low初值
    sta.push(u);                // 将节点u压入栈中
    in
    for(int i = head[u]; ~i ; i = e[i].enxt){
        int v = e[i].v;
```

```
        if( !dfn[v]) {          // 如果节点v未被访问过
            tarjan(v);          // 继续向下找
            low[u] = min(low[u],low[v]);
        }
        else if( instack[v]){  // 反祖边,节点v还在栈内
            low[u] = min(low[u],dfn[v]); //low[u] = min(low[u], low[v]) 理论上这样
写也可以
        }

    }
    if( dfn[u] == low[u]){    // 如果节点u是强连通分量的根
        color_cnt++;
        int t = -1;
        do {
             t = sta.top(); sta.pop();
             instack[t] = 0;  // 将v退栈，为该强连通分量中一个顶点
             color[t] = color_cnt;
        } while( u != v);
    }
}
```

7.3 无向图连通性

7.3.1 图的割点

```
int root; //root点
void tarjan(int u){
    int child=0;//记录root的孩子数
    dfn[u] = low[u] = ++cnt; // 编号

    int i;
    for(i=head[u];i!=-1;i=E[i].next){ //遍历相邻点
        int v = E[i].v ;//另一个点
        if(! dfn[v] ){ //v点的编号为0，也就是没有被访问
            tarjan(v); //从这个点开始dfs
            low[u] = min(low[u],low[v]);//树枝边
            if( u == root) child++; // u是root点

            if(low[v] >= dfn[u] && u != root) // 情况2
                cut[u] =1;
        }
        else //注意:v可能是u的父亲,但没有关系,最多low[u] == dfn[fa[u]]
            low[u] = min(low[u],dfn[v]); //回边
```

```
    }
    // 退出这个点
    if( u == root && child >1)  //情况1:
        cut[u] = 1;
}
```

7.3.2 图的割边

割边：如果在图 G 中删去一条边 e后，图 G的连通分量数增加，即 W (G-e)>W (G)，则称边 u为 G$ 的桥，又称割边或关节边。

性质：对于一条边 $< u, v >$，v 是 u 的孩子如果儿子及儿子的子孙均没有指向 u 的祖先的后向边时，$< u, v >$ 是割边。 $(low[v] > dfn[u])$

```
void CutEdge(int cur,int par)
{    dfn[cur]=low[cur]=++Index;

    for(int i=head[cur];i!=-1;i=e[i].next)
    {
        int v=e[i].v;
        if(v==par)continue; //注意这里和求割点的不同,这里不能用父亲跟新
                            // 但求割点可以用父亲更新,想想为什么!
        if(!dfn[v])
        {
            CutEdge(v,cur);
            if(low[cur]>low[v])
                low[cur]=low[v];
            if(low[v]>dfn[cur])
            {
                    ans[nAns++]=e[i].id;
            }
        }
        else if(low[cur]>dfn[v])
            low[cur]=dfn[v];
    }
}
```

7.3.3 双联通分量

7.3.3.1 点连通分量

点连通图：删去任意一个点之后图依然是连通的.

其他:

一个连通图，如果任意两点至少存在两条点不重复路径，则称这个图为点双连通的.

点连通度

点双连通图的定义等价于任意两条边都同在一个简单环中

对一个无向图，点双连通的极大子图称为点双连通分量（简称双连通分量）

性质

- 如何证明割点在两点双之间呢？

样例

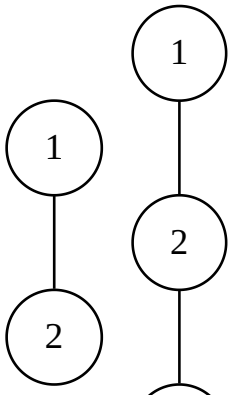


图 1

图 2

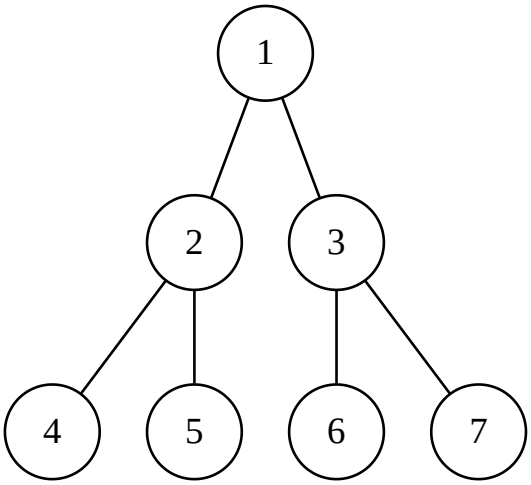


图 3

点连通分量的个数

- 图 1:1
  - $(1, 2)$
- 图 2:2
  - $(1, 2), (2, 3)$
- 图 2:6
  - $(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)$

模板

割点可以属于多个点双连通分量，其余的点和边只属于一个点双连通分量.

对于每两个点双连通分量，最多只有一个共点即割点。任意一个割顶都是至少两个点双连通的公共点.

核心：求解割顶的过程中用一个栈保存遍历过的边（注意不是点！因为不同的双连通分量存在公共点即割顶），之后每当找到一个点双连通分量，即子结点 v 与父节点 u 满足关系 low[v]>=dfn[u]，我们就将栈里的东西拿出来直到遇到当前边.

```
int dfn[maxn],low[maxn];
int idx,bcc_cnt;
stack<int> sta; //栈,存边的编号
//点的颜色,就是点属于哪个bcc
//防止bbc含有一个点多次
int color[maxn];
vector<int> bcc[maxn];//属于某个bcc的点有哪些

void tarjan(int u){
    dfn[u]=low[u] = ++idx;
    int i;
    for(int i=head[u];~i;i=e[i].next){
        int v=  e[i].v;
        if( !dfn[v]){ //没有访问过
            sta.push(i); //存边入栈
            tarjan(v);
            low[u] = min(low[u],low[v]);
            //!!!!注意:
            //这里没有判断u!=root
            if( low[v] >=dfn[u]){
                bcc_cnt++;
                while(1){
                    int i = sta.top(); sta.pop();
                    int uu = e[i].u,vv = e[i].v;
                    if(color[uu] != bcc_cnt){
                        bcc[bcc_cnt].push_back(uu);
                        color[uu] = bcc_cnt;
                    }
                    if(color[vv] != bcc_cnt){
                        bcc[bcc_cnt].push_back(vv);
                        color[vv] = bcc_cnt;
                    }
                    if( uu == u && vv == v) break;
                }
            }
        }
```

```
        else low[u] = min(low[u],dfn[v]);
    }

}
void find_bcc(){
    int i;
    for( i =1;i<=n;i++)
        if( !dfn[i]) tarjan(i);
}

```

7.3.3.2 边连通分量

两遍 dfs:

1. tarjan 求出所有的割边 (桥), 并标记
2. 不走桥的情况下可以遍历的点属于同一个边双连通分量

```
int low[maxn],dfn[maxn];
int col[maxm<<1];//用于标记桥
int ord=1;
void tarjan(int u,int pre){
    low[u]=dfn[u]=ord++;
    int i;
    for(i=head[u];i!=-1;i=e[i].next){
        int v=e[i].v;
        if(v!=pre){
            if(!dfn[v]){
                tarjan(v,u);
                low[u]=min(low[u],low[v]);

                if(low[v]>dfn[u]){
                    col[i]=col[i^1]=1;//标记边及其反向边
                                      //注意: 边的编号从0开始
                }
            }
            else {
                low[u]=min(low[u],dfn[v]);
            }
        }
    }
}

```

```
int mark;
int color[maxn];//不同的边双连通分量会被染为不同的颜色
void dfs(int u,int pre){
    color[u]=mark;

    int i;
    for(i=head[u];i!=-1;i=e[i].next){
        int v=e[i].v;
        if(!col[i] and v!=pre and !color[v]){//该边非桥,不为前驱,且没有被染色
            dfs(v,u);
        }
    }

}

void handle(){
    tarjan(1,0);//原图连通

    int i;
    for(i=1;i<=n;i++){
        if(!color[i]){//该点没有被染过
            mark++;
            dfs(i,0);
        }
    }
}
```

7.3.4 割点，割边，点双，边双四合一模板

```
/*-----------------
* author: Rainboy
* email: rainboylvx@qq.com
* time: 2020年 05月 17日 星期日 18:46:05 CST
* problem:  online_judge-_id
*----------------*/
#include <bits/stdc++.h>
#define For(i,start,end) for(i = start ;i <= end; ++i)
#define Rof(i,start,end) for(i = start ;i >= end; --i)
typedef long long ll;
using namespace std;

/* ======= 全局变量 =======*/
const int maxn = 1e5+5;
```

```
15 const int maxe = 1e6+5;
16 int n,m;
17
18 /* ======= 全局变量 END =======*/
19 /* ========= 向量星 1 =========*/
20 namespace xlx1 {
21     int head[maxn],edge_cnt = 0;
22     struct _e{ int u,v,w,next; }e[maxe];
23     void inline xlx_init(){ edge_cnt = 0; memset(head,-1,sizeof(head)); }
24     void addEdge(int u,int v,int w=0){ e[edge_cnt] = { .u = u,.v=v,.w=w,.next
25 =head[u]}; head[u] = edge_cnt++; }
26     void add(int u,int v,int w=0){ addEdge(u, v,w); addEdge(v, u,w);}
27 }
28 /* ========= 向量星 1 END =========*/
29
30 // ==== 此模板可以求无向图割点,割边,边分,点双
31 #define CUT_N  //割点 开关 要初始化 root
32 #define BRIDGE //割边 开关
33 #define N_BCC  //点双 开关
34 #define E_BCC  //边双 开关
35 //不用加instack
36 namespace UDG_tarjan {
37     using namespace xlx1; // 从0开始存边
38     typedef long long ll;
39     int low[maxn],dfn[maxn];
40     int index=0,bridge= 0,child=0,root,t=-1;
41
42 #ifdef CUT_N
43     bool cut_n[maxn];   //点是否割点
44 #endif
45
46 #ifdef BRIDGE
47     bool cut_e[maxe];   //边是否割边
48 #endif
49
50 #ifdef E_BCC
51     int color_n[maxn],color_n_cnt=0;  //边双 给点染色
52     stack<int> sta_e;   //边双 存点点栈
53 #endif
54
55 #ifdef N_BCC
56     stack<int> sta_n;   //存边的栈
57     int color_e_cnt=0,color_e[maxe]; //对边进行染色
58     int color_n_t[maxn]; //临时
```

```
59      int nbcc_cnt=0; //点双计数
60      vector<int> nbcc_v[maxn];
61  #endif
62
63      void tarjan(int u,int E){
64          low[u] = dfn[u] = ++index;
65  #ifdef E_BCC
66          sta_e.push(u); //边双 存点入栈
67  #endif
68          for(int i= head[u]; ~i ;i=e[i].next){
69              int v = e[i].v;
70              if( !dfn[v]){
71  #ifdef N_BCC
72                  sta_n.push(i); //点双 存边入栈
73  #endif
74  #ifdef CUT_N
75                  if( u == root) child++;   //割点 根点孩子加1
76  #endif
77                  tarjan(v,i);
78                  if( low[u] > low[v]) low[u] = low[v];
79  #ifdef BRIDGE   // 割边
80                  if( low[v] > dfn[u]){ bridge++; cut_e[i]= cut_e[i^1] = 1; }
81  #endif
82  #ifdef CUT_N    // 割点
83                  if( low[v] >= dfn[u] && u != root) cut_n[u] = 1;
84  #endif
85  #ifdef N_BCC
86                  //点双 对边进行染色
87                  if( low[v] >=dfn[u] ){
88                      nbcc_cnt++;
89                      t=-1; do {
90                          t = sta_n.top();sta_n.pop();
91                          color_e[t] = color_e[t^1] = color_e_cnt;
92                          int u = e[t].u,v = e[t].v; //核心思想：不重复的放入
93                          if( color_n_t[u] != nbcc_cnt)
94  nbcc_v[nbcc_cnt].push_back(u),color_n_t[u] = nbcc_cnt;
95                          if( color_n_t[v] != nbcc_cnt)
96  nbcc_v[nbcc_cnt].push_back(v),color_n_t[v] = nbcc_cnt;
97                      }while( t != i );
98                  }
99  #endif
100             }
101             else if( (i^1) != E && low[u] > dfn[v]) low[u] = dfn[v];
102         }
```

```
103 #ifdef CUT_N // 割点
104         if( u == root && child>1) cut_n[u] = 1;
105 #endif
106 #ifdef E_BCC // 边双
107         if( low[u] == dfn[u]){
108             color_n_cnt++,t=-1;
109             do {
110                 t = sta_e.top();sta_e.pop();
111                 color_n[t] = color_n_cnt;
112             }while( t != u);
113         }
114 #endif
115     }
116 }
117
118 int main(){
119     clock_t program_start_clock = clock(); //开始记时
120     //==================
121     xlx1::xlx_init();
122     using namespace xlx1;
123     scanf("%d%d",&n,&m);
124     int i,j;
125     int u,v;
126     For(i,1,m){
127         scanf("%d%d",&u,&v);
128         add(u,v);
129     }
130     UDG_tarjan::root = 1;
131     UDG_tarjan::tarjan(1, -1); //-1 表示没有父子边
132     using namespace UDG_tarjan;
133     printf("bridge %d\n",bridge);
134     printf("每个点的颜色：边双\n");
135     For(i,1,n){
136         printf("%d %d\n",i,color_n[i]);
137     }
138     printf("点是不是割点\n");
139     For(i,1,n){
140         printf("%d %d\n",i,cut_n[i]);
141     }
142     printf("每个点双上的点\n");
143     For(i,1,nbcc_cnt){
144         printf("%d : ",i);
145         for (const auto& e : nbcc_v[i]) {
146             printf("%d ",e);
```

```
        }
        printf("\n");
    }

    //==================
    fprintf(stderr,"\n Total Time: %lf ms",double(clock()-
program_start_clock)/(CLOCKS_PER_SEC / 1000));
    return 0;
}
```

7.4 拓扑排序

7.4.1 拓扑排序

```
/*  kahn拓扑排序
 *
 * */
#include <cstdio>
#include <cstring>

#define N 10000
int n,m;

int indgree[N] = {0}; //入度

int first[N];
int edge_cnt = 0;
struct _e{
    int u,v,w,next;
}e[N];

void addEdge(int u,int v,int w){
    edge_cnt++;
    e[edge_cnt].u = u;
    e[edge_cnt].v= v;
    e[edge_cnt].w=w;
    e[edge_cnt].next = first[u];
    first[u] = edge_cnt;
}

/* 栈的操作 */
int stack[N];
```

```
29 int idx = 0;
30
31 //压栈
32 void push(int x){
33     stack[idx++] = x;
34 }
35
36 //弹出
37 int pop(){
38     return stack[--idx];
39 }
40
41 //栈是否为空
42 bool empty(){
43     return idx == 0;
44 }
45
46 int kahn(){
47     //count用于计算输出的顶点个数
48     int count=0;
49     int i,j,k;
50     //把入度为0的顶点入栈
51     for (i=1;i<=n;i++){
52         if( indgree[i] == 0)
53             push(i);
54     }
55
56     while (!empty()) {//如果栈为空，则结束循环
57         int t = pop();
58         printf("%d ",t); //输出
59         count++;
60
61         // t点周围的点,入度-1
62         for(i=first[t];i!=-1;i = e[i].next){
63             int v = e[i].v;
64             indgree[v]--;
65             if( indgree[v] == 0) //如果入度减少到为0，则入栈
66                 push(v);
67         }
68     }
69
70     return count
71 }
72
```

```
int main(){
    memset(first,-1,sizeof(first));
    scanf("%d%d",&n,&m);
    int i,j,k;
    int t1,t2;
    for(i=1;i<=n;i++){
        scanf("%d%d",&t1,&t2);
        indgree[t2]++;
        addEdge(t1,t2,0);
    }
    kahn();
    return 0;
}
```

DFS 的方法

dfs 搜索本质就是利用栈这种数据结构，那么我们可以用 DFS 来写拓扑排序吗？当然是可以的，想一想，在 DFS 的过程中:

- 边界：一个点没有后趋了，把它存入栈中，
- 一个点回溯了，那这个点后面的点都已经访问过了。把它存入栈中
- 输出中栈中元素

具体的原理：

- 在一个DAG 图的如果一个点的出度为0, 那么这个点的拓扑排序的顺序的一定在最后 (不存在其它的出度为 0 的点), 也在这个点的前趋点的后面.
- 一个点i 如果在DFS 中的要**回溯**了，那这个时候点 i 的后趋点都已经访问了完了，也就是说这个时候点i 的**出度**为 0!!, 先把它存放起来
- 最后把存放的点**倒过来输出**就是拓扑排序

具体看代码

```
/*  dfs拓扑排序
 *
 * */
#include <cstdio>
#include <cstring>

#define N 10000

int first[N];
```

```
int edge_cnt = 0;
struct _e{
    int u,v,w,next;
}e[N];

void addEdge(int u,int v,int w){
    edge_cnt++;
    e[edge_cnt].u = u;
    e[edge_cnt].v= v;
    e[edge_cnt].w=w;
    e[edge_cnt].next = first[u];
    first[u] = edge_cnt;
}

int n,m;
bool instack[N] = {0};

int stack[N];
int stack_index = 0;

void push(int x){
    stack[stack_index++] = x;
}


void topSort_dfs(int u){

    int i;
    for(i=first[u];i!=-1;i=e[i].next){
        int v = e[i].v;
        if( ! instack[v]){ //不在栈中,没有被输出
            topSort_dfs(v);
            //没有后趋
        }
    }
    instack[u] = 1;
    push(u);
}

int main(){
    memset(first,-1,sizeof(first));
    scanf("%d%d",&n,&m);

    int i,j,k;
```

```
    int t1,t2;
    for (i=1;i<=m;i++){
        scanf("%d%d",&t1,&t2);
        addEdge(t1,t2,0);
    }

    for (i=1;i<=n;i++){
        if( ! instack[i]) // 没有在栈中
            topSort_dfs(i);
    }

    for(i=stack_index-1;i>=0;i--)
        printf("%d ",stack[i]);
    return 0;
}
```

7.5 欧拉图与哈密顿图

7.5.1 欧拉回路

核心思想：标记边，用 dfs 的访问的顺序 (栈) 存点.

```
stack<int> sta,ans;
void Euler(int s){
    sta.push(s); //起点入栈
    while( !sta.empty() ){
        int x = sta.top(),i = head[x];
        //找到第一条未访问的边
        while( i!=-1 && vis[i] ) i = e[i].next;
        if( i!=-1){
            sta.push(e[i].v);
            head[x] = e[i].next;
            //标记边,边从0开始编号
            vis[i] = vis[i^1] = 1;
        }
        else { //退出这个点
            sta.pop();
            ans.push(x);
        }
    }
}
```

8 数据结构

8.1 RMQ/ST/ 区间最值

8.1.1 RMQ 区间最值

这样就可以在 $O(nlogn)$ 的时间复杂度内预处理 f 数组:

```
for(i=1;i<=n;i++) f[i][0] = a[i]; //初始化

for(j=1;(1<<j)<=n;j++){ //1<<j 表示处理的范围
    for(i=1;i+(1<<j)-1<=n;i++) // i+(1<<j)-1<=n 表示所求的范围的最后一个值在原数组范围内
        f[i][j] = max (f[i][j-1],f[i+(1<<(j-1))][j-1]);
}
```

$$RMQ(L,R) = max(f[L,x], f[R-2^x+1,x])$$

```
int query(int l,int r){
    //第一种方法
    int x = int( log(r-l+1)/log(2));

    //第二种方法,这种写法比上面的写法慢
    // 例如, luogu3865 就过不了
    int k =0;
    while( (1<<(k+1)) <=(r-l+1) ) k++; // 最后2^{k+1} > r-l+1

    //return max(f[l][k],f[r-(1<<k)+1][k]);
    return max(f[l][x],f[r-(1<<x)+1][x]);
}
```

8.2 树状数组

8.2.1 树状数组：基础

```
#include <cstdio>
#include <cstring>

#define MAX 5000010
int n,m;
int c[MAX] = {0};

```

```
int lowbit(int x){
    return x&(-x);
}

void update(int pos,int num){
    while(pos<=n){
        c[pos] += num;
        pos +=lowbit(pos);
    }
}

//sum(1,pos)
int query(int pos){
    int sum = 0;
    while(pos >0){
        sum+=c[pos];
        pos -= lowbit(pos);
    }
    return sum;
}

int main(){
    scanf("%d%d",&n,&m);
    int i,j,k;
    for (i=1;i<=n;i++){
        scanf("%d",&j);
        update(i,j);
    }

    int t1,t2,t3;
    for (i=1;i<=m;i++){
        scanf("%d%d%d",&t1,&t2,&t3);

        if( t1 == 1){
            update(t2,t3);
        }
        else {
            int ans = query(t3) - query(t2-1);
            printf("%d\n",ans);
        }
    }
    return 0;
}
```

8.2.2 逆序对

求逆序对代码:

```
#include <cstdio>

int n = 5;
int a[]={0,3,1,4,5,2};

int c[100]={0}; //存数状数组

int lowbit(int x){
    return x & (-x);
}

void update(int pos,int num){
    while(pos<=n){  //n代码数组A的长度
        c[pos]+=num;
        pos+=lowbit(pos);
    }
}

int query(int pos){
    int sum = 0;
    while(pos > 0 ){
        sum+=c[pos];
        pos -= lowbit(pos);
    }
    return  sum;
}


int main(){
    int i;
    for(i=1;i<=n;i++){
        update(a[i],1);
        printf("%d ",i-query(a[i])); // 输出前面有几个数比自己大
    }
    return 0;
}
```

8.2.3 区间修改，单点查询

```
const int MAXN = 10000;//最多的点
int a[MAXN];// 原数组
int c[MAXN] = {0};// 树状数组

//原单点更新
void update(int pos,int num){
    while(pos<=MAXN){
        c[pos]+=num;
        pos+=lowbit(pos);
    }
}

//修改一段区间的值
void update_range(int i,int j,int n){
    update(i,n);
    update(j+1,-n);
}
//初始化,形成数状数组
for(i=1;i<=MAXN;i++){
    update(i,a[i]- a[i-1])
}

//查询,也就是单点的值
int query(int pos){
    int sum = 0;
    while( pos > 0 ){
        sum += c[pos];
        pos -= lowbit(pos);
    }
    return sum;
}
```

8.2.4 区间增减 区间查询

```
void update(int pos,int num){
    int t = pos;
    while(pos<=n){  //n代码数组A的长度
        c1[pos]+=num;
        c2[pos] += t*num;
        pos+=lowbit(pos);
    }
}
void update_range(int i,int j,int n){
```

```
    update(i,n);
    update(j+1,-n);
}

int query1(int pos){
    int sum=0;
    while(pos >0){
        sum += c1[pos];
        pos -= lowbit(pos);
    }
    return sum;
}

int query2(int pos){
    int sum=0;
    while(pos >0){
        sum += c2[pos];
        pos -= lowbit(pos);
    }
    return sum;
}

int sum(i,j){
    return (j+1)*query1(j) - query2(j) -i*query1(i-1) +query2(i-1);
}
```

8.2.5 树状数组 区间最值

```
/* pos 位置,v 数值 */
void update(int pos,int v){
    int i,lb;
    c[pos] = a[pos] = v;
    lb = lowbit(pos);
    for(i=1;i<lb;i <<=1){ //利用孩子更新自己
        c[pos] = c[pos] > c[pos-i] ? c[pos] : c[pos-i];
    }
    int pre = c[pos];
    pos+=lowbit(pos);//父亲的位置

    /* 更新父亲 */
    while(pos <= n){
        if( c[pos] < pre){ //更新的父亲
            c[pos] = pre;
```

```
            pos +=lowbit(pos);
        }           //没有更新父亲
        else break;
    }
}
```

8.2.6 二维树状数组

```
void update(int x,int y,int val){
    a[x][y] += val;
    int i,j;
    for(i= x; i <= 横高度; i += lowbit(i) )
        for( j = y ;j <= 纵宽度 ;j += lowbit(j))
            C[i][j] += val;
}
```

```
int query(int x,int y){
    int i,j,sum = 0;
    for(i = x ; i >0 ; i -= lowbit(i) )
        for(j=y;j>0;j-=lowbit(j))
            sum += c[i][j];
    return sum;
}
```

8.2.7 综合模板

```
/* ======== 树状数组 BIT
 * 1.单点增减，区间和
 *    1.1 逆序对
 * 2.区间增减，单点值
 * 3.区间增减，区间和
 * 4.单点修改，末尾压入，区间最值
 * */
namespace bit {
    typedef long long ll;
    ll c[maxn],SIZE=maxn;
    ll c2[maxn]; // c2[i] = i*c[i]
    inline void bit_init(){}

    /* 区间和 */
    inline ll lowbit(ll x) { return x & -x;}
    void update(ll pos,ll add){ while(pos<=SIZE) c[pos]+=add,pos+=lowbit(pos);}
```

```
    /* 差分,区间增减,单点查询*/
    void update_range(ll l,ll r,ll add){ update(l,add);update(r+1,-add); }
    ll query(int pos){ll sum=0;while(pos>0) sum+=c[pos],pos-=lowbit(pos); return
sum;}

    /* 差分,区间增减,区间查询*/
    //核心公式：sum_a[i] = (i+1)×sum_c[i] - sum_{i×c[i]}
    void update_c_c2(ll pos,ll add){ //同时更新c, c2
        ll t = pos; while( pos <= SIZE){ c[pos] += add; c2[pos] += t*add;
pos+=lowbit(pos);}
    }
    void update_range_c_c2(ll l,ll r ,ll add){ update_c_c2(l,
add);update_c_c2(r+1, -add); }
    ll query2(ll pos){ ll sum = 0; while( pos > 0) sum += c2[pos], pos -
=lowbit(pos); return sum; }
    ll query_range_sum(ll l,ll r){ return (r+1)*query(r) - query2(r) - l*query(l-
1) + query2(l-1); }

    // ===== 4.单点修改, 末尾压入, 区间最值
    ll a[maxn]; // 原数组
    void update_by_child(ll pos,ll v){ // alias push
        c[pos] = a[pos] = v;
        ll i,lb = lowbit(pos);
        for(i=1 ; i < lb ; i <<= 1) c[pos] = std::max(c[pos],c[pos-i]);
    }

    void update_ex(ll pos,ll v){
        update_by_child(pos,v); int tmp = c[pos];
        for( pos += lowbit(pos); pos <=SIZE; pos+=lowbit(pos)){
            if( c[pos] < tmp) c[pos] = tmp;
            else break; //没有更新父亲
        }
    }
    ll query_ex(ll l ,ll r){
        ll ex = -1;
        while( l <= r){
            ll left = r - lowbit(r) +1; //范围内的最左点
            if( left >= l) ex = std::max(ex,c[r]) , r = left-1;
            else ex = std::max(ex,a[r]),r--;
        }
        return ex;
    }
}
```

8.3 并查集

8.3.1 并查集

```
namespace BCJ {
    int fa[maxn];
    inline void bcj_init(int x){ for(int i=1;i<=x;i++) fa[i] = i; }
    int find(int x){ if( x == fa[x]) return x; return fa[x] = find(fa[x]); }
    void merge(int x,int y){ fa[find[x]] = find(y); }
}

```

```
void union(int x,int y){
    int rootx = find(x);
    int rooty = find(y);
    if( dep[rootx] <= dep[rooty])
        fa[x] = y, dep[y] = max(dep[y],dep[x]+1);
    else
        fa[x] = x, dep[x] = max(dep[x],dep[y]+1);
}
```

8.4 线段树

8.4.1 单点更新

hdu1166 敌兵布阵 (代码没有提交验证，但是思想正确，如果发现错误，联系我改正)

```
#include <cstdio>

#define lson(rt) (rt<<1)
#define rson(rt) (rt<<1)|1
#define maxn 55555
int tree[maxn<<2];

void pushup(int rt){
    /* 不同的题目有不同的写法 */
    tree[rt] = tree[lson(rt)] +tree[rson(rt)];
}

void build(int l,int r,int rt){
    if(l == r){
        scanf("%d",tree[rt]);//想一想,为什么这样可以读取呢?
        return;
```

```
17      }
18      int m =(l+r)>>1;
19      build(l,m,lson(rt)); //递归建立左子树
20      build(m+1,r,rson(rt));//递归建立右子树
21      pushup(rt);//更新当前点
22  }
23
24
25  void update(int pos,int add,int l,int r,int rt){
26      if(l == r){
27          tree[rt] += add;
28          return;
29      }
30      int m = (l+r)>>1;
31      /* 这样不停的尝试,最的停下的叶子结点一写是poss*/
32      if(pos <=m ) update(pos,add,l,m,lson(rt));
33      else update(pos,add,m+1,r,rson(rt));
34      pushup(rt);
35  }
36
37  int query(int l1,int r1,int l,int r,int rt){
38      if(l1 <= l && r <=r1){
39          return tree[rt];
40      }
41      int m =(l+r)>>1;
42      int ret = 0;
43      if(l1 <=m ) ret+=query(l1,r1,l,m,lson(rt));
44      if(r1 >m ) ret+=query(l1,r1,m+1,r,rsson(rt));
45      return ret;
46  }
47
48  int main(){
49      int T,n;
50      scanf("%d",&T);
51      int i,j,k;
52      for(i=1;i<=T;i++){
53          printf("Case %d:\n",i);
54          scanf("%d",&n);
55          build(1,n,1);
56          char op[10];
57          while(scanf("%s",op)){
58              if(op[0] == 'E') break;
59              int a,b;
60              scanf("%d%d",&a,&b);
```

```
            if(op[0] == 'Q')
                printf("%d\n",query(a,b,1,n,1));
            else if(op[0] == 'S')
                update(a,-b,1,n,1);
            else
                update(a,b,1,n,1);
        }
    }
}
```

8.4.2 成段更新

```
/*================================================================================
* Title : 线段树 成段替换
* Author: Rainboy
* Time  : 2016-05-27 13:05
* update: 2016-05-27 13:05
* ? Copyright 2016 Rainboy. All Rights Reserved.
*================================================================================*/

#include <cstdio>
#include <cstring>
const int maxn = 1000;
int st[maxn<<2];
int flag[maxn<<2];
#define lson(rt) (rt<<1)
#define rson(rt) ((rt<<1)|1)
int n,m;
void pushup(int rt){
    st[rt] = st[lson(rt)] + st[rson(rt)];
}
void pushdown(int rt,int m){
    if(flag[rt]){
        flag[lson(rt)] = flag[rson(rt)] = flag[rt];
        st[lson(rt)] = flag[rt]*(m-(m>>1)); st[rson(rt)] = flag[rt]*(m>>1);
        flag[rt] = 0;
    }
}
void update(int l1,int r1,int c,int l,int r,int rt){
    if(l1 <=l && r<=r1){
        flag[rt] = c; //我们到达一个点
        st[rt] = (r-l+1)*c;
        return ;
```

```
    }
    pushdown(rt,(r-l+1)); //查看当前点对应标记树是不是有标记,如果有就往下压
    int m = (l+r)>>1;
    if( l1 <= m) update(l1,r1,c,l,m,lson(rt));
    if( r1 > m) update(l1,r1,c,m+1,r,rson(rt));
    pushup(rt);
}
int query(int l1,int r1,int l,int r,int rt){
    if(l1<=l && r <= r1){//包含
        return st[rt];
    }

    //路过
    pushdown(rt,(r-l+1));
    int ret = 0;
    int m = (l+r)>>1;
    if(l1 <= m) ret+= query(l1,r1,l,m,lson(rt));
    if(r1 > m ) ret+= query(l1,r1,m+1,r,rson(rt));
    return ret;
}
void build(int l,int r,int rt){
    if( l==r){
        scanf("%d",&st[rt]);
        return ;
    }
    int m = (l+r)>>1;
    build(l,m,lson(rt));
    build(m+1,r,rson(rt));
    pushup(rt);
}
int main(){
    memset(flag,0,sizeof(flag));
    scanf("%d",&n);
    build(1,n,1);
    scanf("%d",&m);
    int i,j,k;
    char c;
    while(m--){
        scanf("%c",&c); //读两次,滤掉\n
        scanf("%c",&c);
        if( c == 'c' ){
            scanf("%d%d%d",&i,&j,&k);
            update(i,j,k,1,n,1);
        }else {
```

```
            scanf("%d%d",&i,&j);
            int ans = query(i,j,1,n,1);
            printf("%d\n",ans);
        }
    }
    return 0;
}

```

8.4.3 线段树合并

1. 动态开点线段树
2. 权值线段树

> 权值线段树能代替平衡树做一些求 k 大、排名、找前驱后继的操作，了解一下就可以啦

合并

```
int merge(int a,int b,int l,int r){
    if(!a) return b;
    if(!b) return a;
    if( l == r){
        // 按题目意思 合并
        // tr[a].val += tr[b].val
        return a;
    }

    int md = (l+r) >> 1;
    tr[a].l = merge(tr[a].l,tr[b].l,l,mid);
    tr[a].r = merge(tr[a].r,tr[b].r,mid+1,r);
    push_up(a);
    return a;
}
```

8.5 「动态开点线段树」 与 「权值线段树」 「感谢」

8.5.1 动态开点线段树

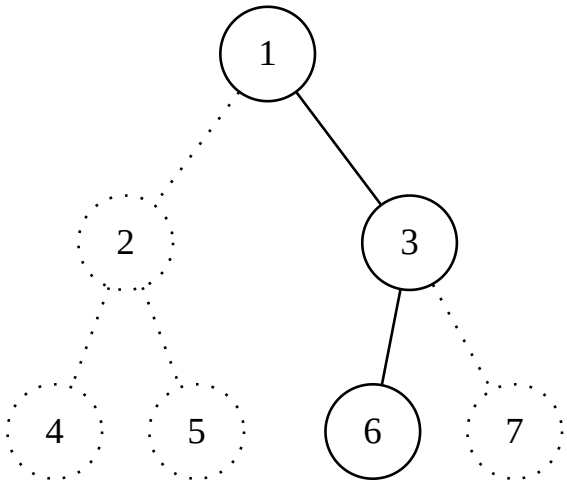
```
int tree[maxn<<2]; //存值
int flag[maxn<<2]; //存懒惰标记
```

我们需要了开 4 倍的空间., 如果用 struct 结构体 左右孩子下标根据计算得出 `lson = rt<<1,rson= (rt<<1)|1;`

```
struct node {
    int val,l,r; //值,左,右孩子的下标
}
```

是不是一定，完整的建立整线段树？

如果出现下面的情况，..., 就不需要完整的建立整棵线段树，这种方式为**动态开点**



操作

变量

0 下标表示空

```
struct { int val,l,r,flag;} tree[maxn];
int cnt ,root;
void newnode (int &p,int val){
    p = ++cnt;
    tree[p].val= val;
    tree[p].l= tree[p].r = 0;
}
```

```
void push_up(int rt){
}
void push_down(int rt){
}
```

```
void update(int u,int L,int R,int x,int val){
}

void update_range(int u,int L,int R,int x)
int query(int u,int L,int R,int l,int r){
}
```

8.6 🚧🔒 主席树

8.6.1 主席树：入门

luogu3834

```
/*-----------------
* author: Rainboy
* email: rainboylvx@qq.com
* time: 2019年 10月 14日 星期一 17:53:55 CST
* problem:  luogu-3834
*----------------*/
#include <bits/stdc++.h>
using namespace std;

const int maxn = 2e5+5; int n,m;

int a[maxn]; //原数组
int b[maxn]; //原数组

int root[maxn];
struct Node {
    int l,r,sum;
};

int cnt=0;
Node tree_node[maxn*40];
inline int get_tree_node(){
    return ++cnt;
}

namespace LSH { //离散化
    int idx=1,i;
    //使用一个数组进行离散化
    int lsh(int arr[],int len){
        for(i=2;i<=len;i++){
```

```
31              if( arr[i] != arr[idx]){
32                  arr[++idx] = arr[i];
33              }
34          }
35          return idx;
36      }
37
38      int get_id(int arr[],int val){
39          return lower_bound(arr+1, arr+idx+1, val)-arr;
40      }
41  }
42
43
44  /* 插入一个点,建立一个线段树 */
45  /* l,r, pre 前一个对应的点,现在这个位置对应的点,插入的值 */
46  void insert(int l,int r,int pre,int &now,int val){
47      /* 复制 */
48      now = get_tree_node();
49      tree_node[now] = tree_node[pre];
50      tree_node[now].sum++;
51      if( l == r ) return;
52       int m = (l+r)>>1;
53       if( val <= m)
54           insert(l, m, tree_node[pre].l, tree_node[now].l, val);
55       else
56           insert(m+1, r, tree_node[pre].r,tree_node[now].r,val);
57  }
58
59  /* 区间查询 */
60  /* l,r 当前结点的区间,L,R 结点的编号 */
61  int query(int l,int r,int L,int R,int k){
62      if( l == r ) return l;
63      int m = (l+r)>>1;
64      // tmp 区间
65      int tmp = tree_node[tree_node[R].l].sum - tree_node[tree_node[L].l].sum;
66      if( k <=tmp) //在左区间
67          return query(l, m, tree_node[L].l, tree_node[R].l, k);
68      else
69          return query(m+1, r, tree_node[L].r,tree_node[R].r,k-tmp);
70  }
71
72  void init(){
73      scanf("%d%d",&n,&m);
74      int i;
```

```
    for(i=1;i<=n;i++){
        scanf("%d",&a[i]);
        b[i] = a[i];

    }
    sort(b+1,b+n+1);
    LSH::lsh(b,n); //离散化
}

int main(){
    init();
    /* 建立树 */
    int i;
    for(i=1;i<=n;i++){
        insert(1, n, root[i-1], root[i], LSH::get_id(b,a[i]) );
    }
    for(i=1;i<=m;i++){
        int l,r,k;
        scanf("%d%d%d",&l,&r,&k);
        int ans = query(1,n,root[l-1],root[r],k);
        printf("%d\n",b[ans]);
    }
    return 0;
}

```

8.7 平衡树

8.7.1 替罪羊树 [Scapegoat Tree]

```
const double alpha  = 0.75;
const double del_alpha = 0.3;
const int base_node = 5;

/* 是否平衡 */
bool imbalence(int now){
    if( max(tzy[tzy[now].l].size , tzy[tzy[now].r].size) > tzy[now].size*alpha +
base_node
    || tzy[now].size - tzy[now].fact > tzy[now].size*del_alpha + base_node)
        return true;
    return false;
}

```

```
14 vector<int> v;
15 /* 中序遍历 */
16 void middle_sort(int now){
17     if(!now ) return;
18     middle_sort(tzy[now].l);
19     if( tzy[now].exist)  v.push_back(now);
20     middle_sort(tzy[now].r);
21 }
22
23 /* 分治重构 */
24 void lift(int l,int r,int &now){
25     if(l == r){ //叶子结点
26         now = v[l];
27         tzy[now].l = tzy[now].r = 0;
28         tzy[now].size = tzy[now].fact = 1;
29         return ;
30     }
31     int m =(l+r)>> 1;
32
33     /* >=key 放右方,见下面[细节] */
34     while( l < m && tzy[v[m]].val == tzy[v[m-1]].val) m--;
35     now=v[m];  //编号
36     /* 如果左区间还存在 */
37     if( l < m ) lift(l, m-1, tzy[now].l);
38     else tzy[now].l = 0;
39     /* 右区间一定存在 */
40     lift(m+1,r,tzy[now].r);
41
42     tzy[now].size = tzy[ tzy[now].l ].size + tzy[ tzy[now].r ].size+1;
43     tzy[now].fact = tzy[ tzy[now].l ].fact + tzy[ tzy[now].r ].fact+1;
44 }
45
46 /* 重建 */
47 void rebuild(int &now){
48     v.clear(); // 清空中序序列
49     middle_sort(now);
50     if( v.empty()){ //中序后,为空
51         now = 0;
52         return;
53     }
54     lift(0,v.size()-1,now);
55 }
56
57 /* 更新一条链 */
```

```
void update(int now,int end){
    if(!now) return;
    if( tzy[end].val < tzy[now].val){
        update(tzy[now].l, end);
    }
    else update(tzy[now].r, end);
    tzy[now].size = tzy[ tzy[now].l ].size + tzy[ tzy[now].r ].size+1;
}

void check(int &now,int end){
    if( now == end) return;
    if( imbalence(now)){
        rebuild(now);
        update(root, now);
        return;
    }
    if( tzy[end].val < tzy[now].val)
        check(tzy[now].l, end);
    else check(tzy[now].r, end);
}
```

8.7.2 Splay 入门

```
/*-----------------
* author: Rainboy
* email: rainboylvx@qq.com
* time: 2019年 11月 17日 星期日 15:26:49 CST problem:  luogu-3369
*----------------*/
#include <bits/stdc++.h>
using namespace std;

/* ======= 全局变量 =======*/
const int maxn = 1e5+5;
int n;
/* ======= 全局变量 END =======*/

/* ========== 快读 ========== */
void in(int &a){
    a = 0;
    int flag = 1;
    char ch = getchar();
    while( ch < '0' || ch > '9'){ if( ch == '-') flag = -1; ch = getchar(); }
    while( ch >= '0' && ch <= '9'){ a = a*10 + ch-'0'; ch = getchar(); }
```

```
    a = a*flag;
}
/* ========== 快读 END ========== */

struct Node
{
    int fa,ch[2],val,cnt,size;  //ch[0]是左儿子, ch[1]是右儿子
}spl[maxn];
int cnt,root; //内存池计数,根编号

//新建节点, 要注意fa指针的初始化
void newnode(int &now,int fa,int val)
{
    spl[now=++cnt].val=val;
    spl[now].fa=fa;
    spl[now].size=spl[now].cnt=1;
}

bool ident(int x,int f){ return spl[f].ch[1] == x; }

void connect(int x,int fa,int ch){
    spl[x].fa = fa;
    spl[fa].ch[ch] = x;
}

inline void push_up(int x){
    spl[x].size=spl[spl[x].ch[0]].size+spl[spl[x].ch[1]].size+spl[x].cnt;
}

void rotate(int x)      //合二为一的旋转
{
    // f:父亲,ff:祖父,k:x是父亲的那个孩子
    int f=spl[x].fa,ff=spl[f].fa,k=ident(x,f);
    connect(spl[x].ch[k^1],f,k);        //x的孩子作为父亲f的孩子
    connect(x,ff,ident(f,ff));          //x作为祖父的孩子
    connect(f,x,k^1);                   //f作为x的孩子
    push_up(f),push_up(x);                //别忘了更新大小信息
}

void splaying(int x,int top)//代表把x转到top的儿子, top为0则转到根结点
{
    // !top => top == 0
    if(!top)
        root=x; //改变根结点
```

```
    while(spl[x].fa!=top) //
    {
        int f=spl[x].fa,ff=spl[f].fa;
        //祖父不是目的结点,要旋转两次,因为有三个点
        if(ff!=top)
            rotate( ident(f,ff)^ident(x,f)? x : f );
            //直线:旋转父亲
            //之字:旋转自己
        rotate(x);      //最后一次都是旋转自己
    }
}

// fa默认为0,now默认为root
void ins(int val,int &now=root,int fa=0)    //递归式, 要传fa指针
{
    if(!now) //当前是空结点,建立,伸展到root
        newnode(now,fa,val),splaying(now,0);
    else if(val<spl[now].val) //去左子树
        ins(val,spl[now].ch[0],now);
    else if(val>spl[now].val) //去右子树
        ins(val,spl[now].ch[1],now);
    else //相等,计数+1,伸展到root
        spl[now].cnt++,splaying(now,0);
}

void delnode(int x)
{
    splaying(x,0);//把x伸展到root
    if(spl[x].cnt>1) spl[x].cnt--;//计数-1
    else if(spl[x].ch[1]) //存在右子树
    {
        int p = spl[x].ch[1];//p 右子树编号
        //p 右子树中的最小值
        while(spl[p].ch[0]) p=spl[p].ch[0];
        splaying(p,x);//p 伸展到 x的孩子

        //x的左孩子,变p的左孩子
        connect(spl[x].ch[0],p,0);
        root=p;//根变p
        spl[p].fa=0;//根的父亲
        push_up(root);//更新size
    } // 只存在左子树,直接做
    else root=spl[x].ch[0],spl[root].fa=0;
}
```

```
109
110 void del(int val,int now=root)
111 {
112     if(val==spl[now].val) delnode(now);
113     else if(val<spl[now].val) del(val,spl[now].ch[0]);
114     else del(val,spl[now].ch[1]);
115 }
116
117 //以下与替罪羊树同
118
119 //得到值val的rank(排名)
120 int getrank(int val)
121 {
122     int now=root,rank=1;
123     while(now)
124     {
125         if(val ==spl[now].val){
126             rank+=spl[spl[now].ch[0]].size;
127             splaying(now,0);
128             break;
129
130         } //去左子树
131         else if(val<spl[now].val)
132             now=spl[now].ch[0];
133         else //去右子树
134         {
135             rank+=spl[spl[now].ch[0]].size+spl[now].cnt;
136             now=spl[now].ch[1];
137         }
138     }
139     return rank;
140 }
141
142 //得到rank(排名)的值
143 int atrank(int rank)
144 {
145     int now=root;
146     while(now)
147     {
148         int lsize = spl[spl[now].ch[0]].size;
149         if( rank >= lsize+1 &&  rank <= lsize+spl[now].cnt){
150             splaying(now, 0);
151             break;
152         }
```

```
153
154         else if(lsize>=rank) // 在左边
155             now=spl[now].ch[0];
156         else // 在右边
157         {
158             rank-= lsize+ spl[now].cnt;
159             now=spl[now].ch[1];
160         }
161     }
162     return spl[now].val;
163 }
164
165
166 int main(){
167     clock_t program_start_clock = clock(); //开始记时
168     //==================
169     in(n);
170     int opt,x;
171     while(n--){
172         in(opt),in(x);
173         switch(opt){
174             case 1: //insert
175                 ins(x);
176                 break;
177             case 2: //del
178                 del(x);
179                 break;
180             case 3: //getrank
181                 printf("%d\n",getrank(x));
182                 break;
183             case 4: //atrank
184                 printf("%d\n",atrank(x));
185                 break;
186             case 5: //pre
187                 printf("%d\n",atrank(getrank(x)-1));
188                 break;
189             case 6: //nxt
190                 printf("%d\n", atrank(getrank(x+1)));
191                 break;
192         }
193     }
194
195
196     //==================
```

```
    fprintf(stderr,"\n Total Time: %lf ms",double(clock()-
program_start_clock)/(CLOCKS_PER_SEC / 1000));
    return 0;
}
```

8.7.3 Treap

```
/* Author:Rainboy 2018-09-08 00:32 */
#include <cstdio>
#include <cstring>

#define N 100005
#define ls tr[p].l      //左孩子
#define rs tr[p].r      //右孩子
const int INF = 0x7fffff7f;

int n;

struct node{
    int l,r,val; //左右孩子,点的值
    int size,rand,cnt;//子树的大小,随机值,该结点出现的次数
} tr[N];
int sz = 0; //编号用

int rmax(int a,int b){
    if(a > b ) return a;
    return b;
}
int rmin(int a,int b){
    if(a < b ) return a;
    return b;
}



inline int rand ( )  {
    static int seed = 733;
    return seed = ( int ) seed * 482711LL % 2147483647;
}

/* 更新当前点的size */
inline void update(int p){
```

```
36      tr[p].size = tr[ls].size + tr[rs].size +tr[p].cnt;
37  }
38
39  void lturn(int &p){
40      int t = tr[p].r;
41      tr[p].r = tr[t].l;
42      tr[t].l= p;
43      tr[t].size = tr[p].size; update(p); p =t; //改变根结点
44  }
45
46  void rturn(int &p){
47      int t = tr[p].l;
48      tr[p].l = tr[t].r;
49      tr[t].r = p;
50      tr[t].size = tr[p].size;
51      update(p);p =t;
52  }
53
54  /* 插入 */
55  void insert(int &p,int x){
56      if( p == 0){ //边界 来到一个空点
57          p = ++sz;
58          tr[p].size = tr[p].cnt= 1;
59          tr[p].val =x;tr[p].rand = rand();
60          return;
61      }
62
63      tr[p].size++; //路过,所以要++
64      if(tr[p].val == x) tr[p].cnt++; //来到一个相同点
65      else if( x > tr[p].val){ //比当前点大,进入右子树
66          insert(rs,x);
67          /* 回溯 */
68          if(tr[rs].rand < tr[p].rand) lturn(p);
69      }
70      else { //进入左子树
71          insert(ls,x);
72          //回溯
73          if( tr[ls].rand < tr[p].rand) rturn(p);
74      }
75  }
76
77  /* 删除 */
78  void del(int &p,int x)
79  {
```

```
    if (p==0) return;
    if (tr[p].val==x)
    {
        if (tr[p].cnt>1) tr[p].cnt--,tr[p].size--;//如果有多个直接减一即可。
        else
        {
            if (ls==0||rs==0) p=ls+rs;//单节点或者空的话直接儿子移上来或者删去即
可。
            else if (tr[ls].rand<tr[rs].rand) rturn(p),del(p,x);
            else lturn(p),del(p,x);
        }
    }
    else if (x>tr[p].val) tr[p].size--,del(rs,x);
    else tr[p].size--,del(ls,x);
}

/* 找到排名 ,所有比x点小的点有多少个*/
int find_pm(int p,int x){
    if(p==0) return 0;
    if(tr[p].val == x) return tr[ls].size+1;
    if(x > tr[p].val ) //x比当前点大,进入右子树
        return tr[ls].size+tr[p].cnt+find_pm(rs,x);
    else        //x比当前点要小于
        return find_pm(ls,x);
}

/* 查询排名为x的数 */
int find_sz(int p,int x){
    if(p ==0 ) return 0;
    if( x <=tr[ls].size)
        return find_sz(ls,x);

    x -= tr[ls].size;
    if(x<=tr[p].cnt ) return tr[p].val;
    x -= tr[p].cnt;
    return find_sz(rs,x);
}

/* 找到前趋 */
int find_qq(int p,int x){
    if(p == 0 ) return -INF;
    if( tr[p].val <x )
        return rmax(tr[p].val,find_qq(rs,x));
    else
```

```
        return find_qq(ls,x);
}
/* 找到后继 */
int find_hj(int p,int x){
    if(p == 0 ) return INF;
    if( tr[p].val <= x)
        return find_hj(rs,x);
    else
        return rmin(tr[p].val,find_hj(ls,x));
}

int main(){
    scanf("%d",&n);

    int i,flag,x,rt=0;
    for (i=1;i<=n;i++){
        scanf("%d%d",&flag,&x);
        if( flag == 1)
            insert(rt,x);
        else if( flag == 2)
            del(rt,x);
        else if( flag == 3)
            printf("%d\n",find_pm(rt,x));
        else if( flag == 4)
            printf("%d\n",find_sz(rt,x));
        else if( flag == 5)
            printf("%d\n",find_qq(rt,x));
        else if( flag == 6)
            printf("%d\n",find_hj(rt,x));
    }
    return 0;
}
```

8.7.4 fhq-treap

分裂 (split), 按值 val 分裂成两棵树，x 树，y 树；其中 x 树中的所有值都小于等于 val,y 树中的值都大意 val

root 为当前 dfs 树中子树的根

- 如果 root 点的 val≤给定值，则 root 和其左子树上的点都leq 给定值，那么它们应该属于x 树
  - 对 root 的右子树进行分裂 (split)
  - 分裂后的 x 树为原 x 树的右子树

    - 分裂后的 y 树为原 y 树的左子树
- 如果 root 点的 val> 给定值，则 root 和其右子树上的点都> 给定值，那么它们应该属于Y 树
    - 对 root 的左子树进行分裂 (split)
    - 分裂后的 x 树为原 x 树的右子树
    - 分裂后的 y 树为原 y 树的左子树

```
void split(int now,int val,int &x,int &y){
    if( !now ) {
        x = y = 0;
        return;
    }

    if( fhq[now].val <= val){
        x= now;
        split(fhq[now].r, val, fhq[now].r, y);
    }
    else { // >
        y = now;
        split(fhq[now].l, val, x, fhq[now].l);
    }
    update(now);
}
```

8.8 link cut tree(LCT)

8.8.1 「LCT 入门」序

```
const int maxn = 1e5+5;
struct Node
{
    int fa,ch[2],val,res;   //res是异或结果
    bool flag;               //翻转懒标记
}spl[maxn];
//因为被毒瘤数据卡得TLE一个点，所以全部换成了#define。都是字面意思
#define ls(x) (spl[x].ch[0])
#define rs(x) (spl[x].ch[1])
#define fa(x) (spl[x].fa)
#define ident(x,f) (rs(f)==x)       //和下面的connect都是Splay的辅助函数
#define connect(x,f,s) spl[fa(x)=f].ch[s]=x
#define update(x) spl[x].res=spl[ls(x)].res^spl[rs(x)].res^spl[x].val
#define ntroot(x) (ls(fa(x))==x||rs(fa(x))==x)  //判断结点是否为Splay的根
#define reverse(x) std::swap(ls(x),rs(x)),spl[x].flag^=1
```

9 其它算法

9.1 快读

简短压行版

```
int read(){
    int x=0,t=1;
    char ch=getchar();
    while(ch<'0' || ch>'9'){ if(ch=='-') t=-1; ch=getchar(); }
    while(ch<='9' && ch>='0') { x=x*10+ch-'0'; ch=getchar(); }
    return x*t;
}
```

更快的快读与快输 (补充 by Rainboy)

```
#include <iostream>
#include <ctime>
#include <cstdio>
#include <cctype>
namespace FastIO        //使用命名空间
{
    char buf[1 << 21], buf2[1 << 21], a[20], *p1 = buf, *p2 = buf, hh = '\n';
    int p, p3 = -1;
    inline int getc()
    {
        // *p1++
        return p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, 1 << 21, stdin), p1 ==
p2) ? EOF : *p1++;
    }
    inline int read()
    {
        int ret = 0, f = 0;
        char ch = getc();
        while (!isdigit(ch))
        {
            if (ch == '-')
                f = 1;
            ch = getc();
        }
        while (isdigit(ch))
        {
```

```
            ret = ret * 10 + ch - 48;
            ch = getc();
        }
        return f ? -ret : ret;
    }
    inline void flush()
    {
        fwrite(buf2, 1, p3 + 1, stdout), p3 = -1;
    }
    inline void print(int x)
    {
        if (p3 > 1 << 20)
            flush();
        if (x < 0)
            buf2[++p3] = 45, x = -x;
        do
        {
            a[++p] = x % 10 + 48;
        } while (x /= 10);
        do
        {
            buf2[++p3] = a[p];
        } while (--p);
        buf2[++p3] = hh;
    }
}
#define read() FastIO::read()
#define print(x) FastIO::print(x)
```

9.2 序列中和不超过 K 的对数

核心思想:

- 与最小数配对
- 不停缩小区间

```
    sort(a+1, a+1+n);

    int l=1,r=n;
    int ans = 0;

    while( l < r){
        if(a[l] + a[r] <= k){
```

```
            ans += r - l;
            l++;
        }
        else
            r--;
    }
```

9.3 离散化

```
#include <bits/stdc++.h>
using namespace std;

int a[100];
int b[100];
int n;

int main(){
    int i,j;
    for (i=1;i<=n;i++){
        scanf("%d",&a[i]);
        b[i] = a[i];
    }
    sort(b+1,b+n+1);
    int nn = unique(b+1,b+n+1) - b -1;
    for(i = 1;i<=n;i++){
        a[i] = lower_bound(b+1,b+n+1,a[i]) - b;
    }
    return 0;
}
```