

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 Floyd-Warshall 算法	35
7.1.1.1 floyd	35
7.1.1.2 floyd 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

```

1 void bubble_sort(int a[],int n){
2     int i,j;
3     int tmp;
4     for(i=1;i<=n-1;i++) //n个数,要进行n-1趟排序
5         for(j=1;j<=n-i;j++){//第i趟排序的最后一个下标:n-i
6             if(a[j] > a[j+1]){
7                 tmp =a[j];
8                 a[j] =a[j+1];
9                 a[j+1]=tmp;
10            }
11        }
12 }

```

```

1 //冒泡排序
2 for (i=1;i<=n-1;i++){ //n-1轮排序
3     for(j=1;j<=n-i;j++){
4         if( a[j] > a[j+1]){
5             cnt++;
6             swap(a[j],a[j+1]);
7         }
8     }
9 }

```

1.2 快速排序

```
1 void quicksort(int l, int r) {
2     int i, j, t, tmp;
3     if(l > r) return;
4     tmp = a[l]; //tmp中存的就是基准数
5     i = l, j = r;
6     while(i != j) { //顺序很重要, 要先从右边开始找
7         while(a[j] >= tmp && i < j) j--;
8         //再找左边的
9         while(a[i] <= tmp && i < j) i++;
10        //交换两个数在数组中的位置
11        if(i < j) swap(a[i],a[j]);
12    }
13    //最终将基准数归位
14    a[l] = a[i];
15    a[i] = tmp;
16    quicksort(l, i-1); //继续处理左边的, 这里是一个递归的过程
17    quicksort(i+1, r); //继续处理右边的, 这里是一个递归的过程
18 }
```

1.3 归并排序

```
1 void merge_sort(int s, int t){
2     //s = start t = T
3     int mid, i, j, k;
4
5     if(s == t) return ; //如果区间只有一个数, 就返回
6
7     mid = (s+t) >> 1; //取中间的点
8     merge_sort(s, mid);
9     merge_sort(mid+1, t);
10
11    i = s, k = s, j = mid+1;
12
13    while(i <= mid && j <= t){
14        if(a[i] <= a[j]){
15            tmp[k] = a[i]; k++; i++;
16        } else {
17            tmp[k] = a[j]; j++; k++;
18        }
19    }
20 }
```

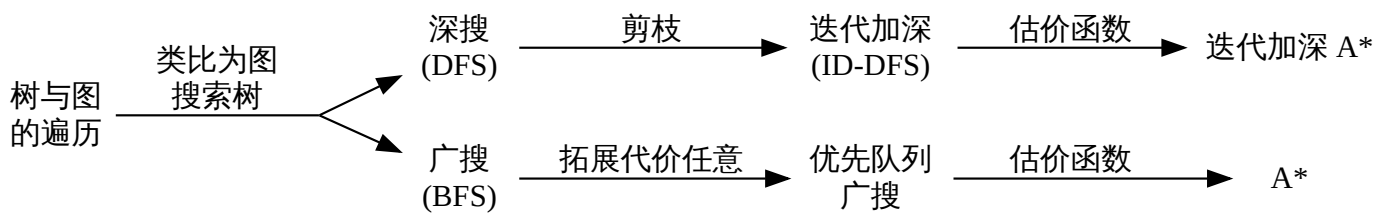
```

21     while(i<=mid) { tmp[k]=a[i];k++;i++;}
22     while(j<=t)    { tmp[k]=a[j];k++;j++;}
23
24     for(i=s;i<=t;i++)
25         a[i]=tmp[i];
26 }

```

2 搜索算法

2.1 序



2.2 深度优先 : DFS

2.2.1 深度优先搜索

```

1 void dfs( state arguments){
2
3     if(到达边界条件)
4         结束函数
5
6     for(i=0;i<=max;i++){
7
8         state_next = opt
9         //
10
11         dfs(state_next)
12
13     }
14 }

```

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

```

1 void _01bfs(int state,int sx,int sy){
2     /* 清空标记 */
3     memset(vis,0,sizeof(vis));
4     typedef struct { int x,y,step; } node;
5     deque<node> q;
6     q.push_back({sx,sy,0}); //加入起点
7
8     while( !q.empty()){
9         node h = q.front();
10        q.pop_front();
11        if( vis[h.x][h.y])
12            continue;
13        vis[h.x][h.y] = 1; //标记
14
15        dis[state][h.x][h.y] = h.step;
16
17        int i;
18        for(i = 0; i < 4;i++){
19            int nx = h.x + fx[i][0];
20            int ny = h.y + fx[i][1];
21            if( in_map(nx,ny) && !vis[nx][ny] && _map[x][y] != '#' ){
22                if( _map[x][y] == '0')
23                    q.push_front({nx,ny,h.step});
24                else
25                    q.push_back({nx,ny,h.step+1})
26            }
27        }
28    }
29 }
30

```

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。

```

1 //F = g + h 估价函数 g实际 h估计
2 struct node {
3     int g,h;
4     friend bool operator<(const node &a,const node &b){
5         return a.g+a.h > b.g+b.h;
6     }
7 };
8 bool astart(){
9     node start;
10    priority_queue<node> q;
11    q.push(start);
12
13    while( !q.empty() ){
14        node head = q.top(); q.pop();
15        if( head is 终点) return 1;
16        if( head in vis ) continue;
17        vis[head] = 1;
18
19        for( each v in <head,v> ){
20            if( v in vis) continue;
21            node t = get_f(v); 计算v在F值
22            q.push(t); //放入优先队列

```

```

23     }
24
25 }
26 return -1; //没有找到目标
27 }
28

```

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

```

1 bool iddfs(int d,int maxd, args...){
2     if( d == maxd+1){
3         if( 符合题意 ) return true;
4     }
5
6     for(int abble = 1; abble <= 最大次数; abble++ ){
7         if( 估价函数(d,maxd) 符合条件 )
8             iddfs(d+1,maxd, args...)
9     }
10 }
11
12 //迭代加深
13 for( int maxd = 1 ; ; maxd++){
14     if(iddfs(dep=1,maxd, ... ) ) break;
15 }

```

3 字符串算法

3.1 字符串 hash

```

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

```

```

9         for(int i=1;i<=maxn;i++) b[i] = b[i-1]*seed;
10    }
11    void get_sum(char s[],int len){
12        sum[0] = 0;
13        for(int i=1;i<=len;i++) sum[i] = sum[i-1]*seed + (ULL)(s[i]-start+1);
14    }
15
16    ULL get_hash(char s[],int len){
17        ULL s = 0;
18        s = s*seed + (ULL)(s[i]-start+1);
19    }
20
21    ULL get_hash_range_from_sum(int l,int r){
22        return sum[r] - sum[l-1]*b[r-l+1];
23    }
24 }
25

```

3.2 kmp 算法

3.2.1 KMP 算法

```

1  const int kmp_max_len = 1e5;
2  struct KMP{
3      int la,lb;
4      char a[kmp_max_len],b[kmp_max_len];
5      int next[kmp_max_len];
6      int cnt =0; //匹配字符串的数量
7
8      void deal_next(){
9          int i,j=0;
10         next[1] = 0;
11         for(i=2;i<=lb;++i){
12             while(j && b[j+1] != b[i]) j = next[j];
13             if( b[j+1] == b[i]) j++;
14             next[i] = j;
15         }
16     }
17     void kmp(){
18         int i,j=0;
19         for(i=1;i<=la;++i){
20             while(j && b[j+1] != a[i]) j = next[j];
21             if( b[j+1] == a[i]) j++;

```

```

22         if( j == lb){
23             //printf("%d\n",i-lb+1);
24             cnt++;
25             j=next[j];
26         }
27     }
28 }
29 };

```

3.3 字典树 / Trie

3.3.1 字典树

```

1 struct trie {
2     int nex[100000][26], cnt;
3     bool exist[100000]; // 该结点结尾的字符串是否存在
4
5     void insert(char *s, int l) { // 插入字符串
6         int p = 0;
7         for (int i = 0; i < l; i++) {
8             int c = s[i] - 'a';
9             if (!nex[p][c]) nex[p][c] = ++cnt; // 如果没有, 就添加结点
10            p = nex[p][c];
11        }
12        exist[p] = 1;
13    }
14    bool find(char *s, int l) { // 查找字符串
15        int p = 0;
16        for (int i = 0; i < l; i++) {
17            int c = s[i] - 'a';
18            if (!nex[p][c]) return 0;
19            p = nex[p][c];
20        }
21        return exist[p];
22    }
23 };

```

3.4 AC 自动机

```

1 namespace AC {
2
3     //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;
```

```
48         k = next[k];
49     }
50     u = ch[u][c];
51 }
52 }
53 }
54
```

4 分治算法

4.1 二分查找

4.1.1 二分查找

```
1  #include <stdio>
2
3  int a[] = {1,2,3,4,5};
4
5  int bsearch(int l,int r,int key){
6
7      while (l<=r){
8          int m = (l+r)>>1;
9          if(a[m] == key)
10             return m;
11          if( key < a[m])
12             r = m-1;
13          else
14             l = m+1;
15      }
16      return -1;
17 }
18
19 int main(){
20     int len_a = sizeof(a)/sizeof(a[0]);
21
22     int ret = bsearch(0,len_a-1,2);
23     printf("%d\n",ret);
24     return 0;
25 }
```

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

```

1  #include <stdio>
2
3  int a[1000];
4  int n;
5  int f[1000];
6
7  int main(){
8      scanf("%d",&n);
9      int i;
10     int j;
11     for (i=1;i<=n;i++){
12         scanf("%d",&a[i]);
13     }
14
15     //每个点的f值不可能小于1
16     for(i=1;i<=n;i++) f[i] =1;
17
18     int max = 1;//这里是1,想想为什么
19     for (i=2;i<=n;i++){
20         for(j=1;j<i;j++){
21             if( a[j] <= a[i] && f[i] < f[j]+1){
22                 f[i] = f[j]+1;
23                 if(max < f[i])
24                     max = f[i];
25             }
26         }
27     }
28
29     printf("%d",max);
30
31     return 0;
32 }

```

```

1  a: 7 2 9 3 4 10 6 1  原数组
2  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：

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

```
1 // 求最长不下降子序列
2 #include <cstdio>
3 #include <cstring>
4
5 int n;
6 int a[100];
7 int c[100],f[100];
8
9 //[l,r)
10 int upper_bound(int l,int r,int key){
11     while(l < r){
12         int m = (l+r)>>1;
13         if( c[m] <= key)
14             l = m+1;
15         else
16             r = m;
17     }
18     return l;
19 }
20
21 void init(){
22     memset(c,0x7f,sizeof(c));
```

```
23     scanf("%d",&n);
24     int i;
25     for (i=1;i<=n;i++){
26         scanf("%d",&a[i]);
27     }
28 }
29
30 void lis(){
31     int i,j,k;
32     c[1] = a[1];
33     f[1] = 1;
34     for(i=2;i<=n;i++){
35         int pos = upper_bound(1,i,a[i]);
36         pos--;
37         f[i] = pos+1;
38         if( c[ f[i] ] > a[i])
39             c[ f[i] ] = a[i];
40     }
41 }
42 int main(){
43     init();
44     lis();
45     int i;
46     for (i=1;i<=n;i++){
47         printf("%d %d\n",i,f[i]);
48     }
49
50     return 0;
51 }
```

5.1.2 LCS 最长公共子序列

5.1.2.1 LCS 最长公共子序列

```
1 #include <cstdio>
2 #include <cstring>
3
4 char s1[5010];
5 char s2[5010];
6 int l1,l2;
7
8 int f[5010][5010] = {0};
9
```

```
10 int max(int a,int b){
11     if( a> b)
12         return a;
13     return b;
14 }
15
16 int lcs(){
17     int i,j;
18
19     for(i=1;i<=l1;i++)
20         for(j=1;j<=l2;j++){
21             if( s1[i] == s2[j]){
22                 f[i][j] = f[i-1][j-1]+1;
23             }
24             else
25                 f[i][j] = max(f[i][j-1],f[i-1][j]);
26         }
27     return f[l1][l2];
28 }
29 int main(){
30     scanf("%s",s1+1);
31     scanf("%s",&s2[1]);
32     l1 = strlen(s1+1);//求长度
33     l2 = strlen(s2+1);
34
35     int m = lcs();
36     printf("%d",m);
37
38     return 0;
39 }
```

		j	0	1	2	3	4	5	6
		y _j		B	D	C	A	B	A
i	x _i								
0			0	0	0	0	0	0	0
1	A		0	0	0	0	1	←1	1
2	B		0	1	←1	←1	1	1	←2
3	C		0	1	1	2	←2	2	2
4	B		0	1	1	2	2	3	←3
5	D		0	1	2	2	2	3	3
6	A		0	1	2	2	3	3	4
7	B		0	1	2	2	3	4	4

LCS 转 LIS

使用，求下面的两个序列的 LCS

1 | 2 1 3

2 | 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

1 | for i=1->N //前i个物品

2 | for j=C->w[i] //容量从大到小

3 | f[j] = max(f[j],f[j-w[i]]+v[i])

5.2.2 完全背包

1 | for i = 1->N //枚举前i个物品

2 | for v=c[i]->V //枚举背包容量

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

5.2.3 多重背包

```
1 |         for (int k=1; k<=c; k<=1) { //<<右移 相当于乘二
2 |             value[count] = k*v;
3 |             size[count++] = k*s;
4 |             c -= k;
5 |         }
6 |         if (c > 0) {
7 |             value[count] = c*v;
8 |             size[count++] = c*s;
9 |         }
```

5.2.4 分组背包

```
1 | int i,j,k;
2 | for(i=1;i<=N;i++)
3 |     for(K=C;k>=0 ;k--){ //先枚举容量
4 |         for(j=1;j<=num[i];j++){ //再枚举组内物品
5 |             if(W[i][j] > k) continue;
6 |             f[k] = max{
7 |                 f[k],
8 |                 f[k-W[i][j]] + V[i][j]
9 |             }
10 |        }
11 |    }
```

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

```
1 | namespace Trajan_lca {
2 |
```

```

3   int qhead[maxn],q_cnt = 0;
4   struct Query { int u,v,next,id; }query[maxe];
5   void query_queue_init(){   q_cnt = 0; memset(qhead,-1,sizeof(qhead)); }
6   void addQuery(int u,int v){ query[q_cnt]={u,v,head[u],q_cnt};
7   qhead[u]=q_cnt++; }
8
9   int fa[maxn],vis[maxn];
10  void find(int u){ //路径压缩
11      if( fa[u] == u) return u;
12      return fa[u] = find(fa[u]);
13  }
14
15  void tarjan_lca(int u){
16      fa[u] = u;
17      for(int i = head[u]; ~i ;i = e[i].next){
18          int v = e[i].v;
19          tarjan_lca(v);
20          fa[v] = u;
21      }
22      vis[u] = 1; //标记 放这里的原因就是 求lca(6,6)这种点
23      for(int i = qhead[u] ;~i ; i =query[i].next){
24          int v = query[i].v;
25          if( vis[v]) ans[query[i].id] = find(v);
26      }
27  }
28 }
29

```

6.1.2 LCA 在线算法 - 树上倍增

```

1  //===== 树上倍增 BEG
2  namespace BZ_LCA {
3      const int SIZ = 35;
4      int f[maxn][SIZ+1],len[maxn][SIZ+1],dep[maxn];
5      using namespace xlx1;
6      void dfs_init(int u ,int d,int fa,int w){
7          dep[u] = d; f[u][0] = fa; len[u][0] = w;
8          for(int i= 1;i<=SIZ;++i){
9              f[u][i] = f[f[u][i-1]][i-1];
10             len[u][i] = len[u][i-1]+len[f[u][i-1]][i-1];
11         }
12         for(int i=head[u];~i;i=e[i].next){

```

```

13         int v = e[i].v , w = e[i].w;
14         if( v == fa) continue;
15         dfs_init(v,d+1,u,w);
16     }
17 }
18 int find_lca(int u,int v){
19     int sum = 0;
20     if( dep[u] < dep[v]) swap(u,v); //保证u点的深度深
21     for(int i=SIZ;i>=0;--i){
22         if( dep[f[u][i]] < dep[v]) continue; //不跳的区域
23         sum+=len[u][i];
24         u = f[u][i];
25     }
26     if(u == v) return sum;
27     for(int i=SIZ;i>=0;--i){
28         if( f[u][i] == f[v][i] ) continue;
29         sum+=len[u][i];
30         sum+=len[v][i];
31         u = f[u][i];
32         v = f[v][i];
33     }
34     return sum+len[u][0]+len[v][0];
35 }
36 }
37 //===== 树上倍增 END
38

```

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

```

1 // 向量星 略
2
3 /* ===== 并查集 ===== */
4 const int maxn = 1e5+5;
5 int fa[maxn];
6 //并查集 初始化
7 inline void bcj_init(int x){ for(int i=1;i<=x;i++) fa[i] = i; }

```

```

8 //查找 and 路径压缩
9 int find(int x){
10     if( x == fa[x]) return x;
11     return fa[x] = find(fa[x]);
12 }
13 /* ===== 并查集 end ===== */
14
15 bool cmp( edge a,edge b) {return a.w < b.w;}
16 // 传入点数, 返回mst的值, 不连通返回-1
17 int Kruskal(int n){
18     bcj_init(n); //初始化
19     sort(e+1,e+1+edge_cnt,cmp); //对边从小到大排序
20     int ans = 0,cnt = 0; //答案, 选边的数量
21
22     for(int i = 1; i<=edge_cnt ; ++i){
23         int u =e[i].u, v =e[i].v, w =e[i].w;
24         int f1 =find(u),f2 = find(v);
25         if( f1 != f2){ //不再同一个集合
26             ans+=w;
27             cnt++;
28             fa[f2] = f1;
29         }
30         if( cnt == n-1) break;
31     }
32     if( cnt < n -1) return -1;
33     return ans;
34 }

```

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 $\langle u, v \rangle$, 加入到 mst 上, 得到小的生成树的值:

$$new_value = mst_value + w(u, v) - maxd(u, v)$$

```

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

```



```

4  }e[maxm<<1];
5
6  //G[i] 表示集合i上的点
7  vector<int> G[maxn];
8
9  long long second_mst(){
10     long long mst,second_mst=0x7fffffffffffffff;
11     int i,j,k=0;
12     /* 初始化 并查集 */
13     for (i=1;i<=n;i++){
14         fa_bcj[i] = i;
15         G[i].push_back(i);
16     }
17
18     for(i=1;i<=m;i++){
19         int f1 = find(e[i].u);
20         int f2 = find(e[i].v);
21         if( f1 != f2){
22             mst += e[i].w;
23             fa_bcj[f1] = f2;
24             e[i].vis = 1; // 这条边在 MST 上
25
26             /* 求maxd*/
27             long long &u = e[i].u,&v = e[i].v,&w = e[i].w;
28
29             for( auto x1 : G[f1]){
30                 for( auto x2 : G[f2]){
31                     maxd[x1][x2] = maxd[x2][x1] = w;
32                 }
33             }
34
35             /* 把集和 f1 合并到f2 上,不再关心f1集合*/
36             G[f2].insert( G[f2].end(),G[f1].begin(),G[f1].end());
37
38             k++;
39             if( k == n-1) break; //选n-1条边
40         }
41     }
42
43     for(i=1;i<=m;i++){
44         if( !e[i].vis){
45             long long t = mst + e[i].w - maxd[e[i].u][e[i].v];
46             second_mst = min(second_mst,t);
47         }

```

```

48     }
49     return second_mst;
50 }
51

```

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。

```

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

```

```

17     }
18 }
19 if(n-siz[x]>n/2) flag=false;//判断节点x是不是重心。
20 if(flag && x<CenterOfGravity) CenterOfGravity=x,father=u;//这里写
21 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

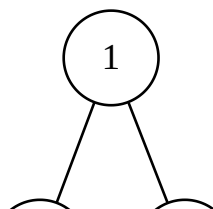
```

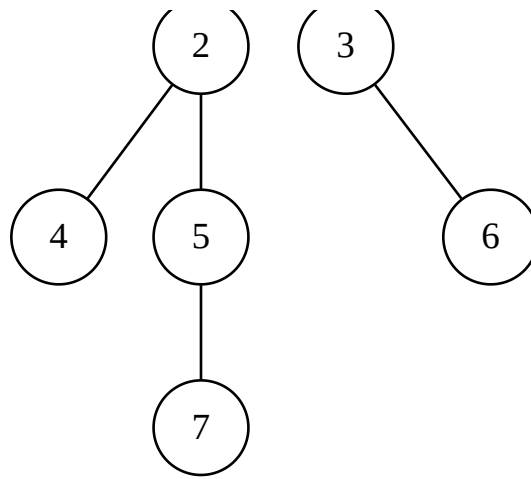
1 //TLP:tree_longest_path
2 namespace TLP {
3     using namespace xlx1;
4     typedef long long ll;
5     ll node,dis;
6     //len: 根到点u的距离
7     void dfs(int u,int fa,int len){
8         if(dis < len ) node = u, dis = len;
9         for(int i = head[u];i!=-1;i=e[i].next){
10             int v = e[i].v;
11             if( v == fa) continue;
12             dfs(v,u,len+e[i].w);
13         }
14     }
15     void work(){
16         dis=-1;dfs(1,0,0);    //第1次dfs
17         dis=-1;dfs(node,0,0);//第2次dfs
18         //得到最长的直径dis
19     }
20 }

```

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
---	---	---	---	---	---	---

```

1  int start[maxn],end[maxn];
2  int dep[maxn],fa[maxn];
3  int dfs_clock = 0;
4  int dfs(int d,int u,int pre){
5      dep[u] = d;
6      fa[u] = pre;
7      start[u] = ++dfs_clock;
8      int i;
9      for(i = head[u]; ~i ; i = e[i].next){
10         int v= e[i].v;
11         if( v != pre){
12             dfs(d+1,v,u);
13         }
14     }
15     end[u] = dfs_clock;
16 }

1  const int maxn = 1e5+5;
2  int in[maxn],out[maxn];
3  inline bool isChild(int a,int b)
4      if( start[b]<= start[a] && end[a] <= end[b])
5          return 1;
6      return 0;
7  }
  
```

6.5 树链剖分

6.5.1 lca

```
1  #include <cstdio>
2  #include <cstring>
3
4  #define maxn 100
5
6  int m,n;
7  int root;
8
9  int head[maxn];
10 struct edge {
11     int next;
12     int v;
13 }E[maxn<<1]; //存两遍边
14 int cnt = 0;
15
16 void addedge(int x,int y){
17     cnt++;
18     E[cnt].v = y;
19     E[cnt].next = head[x];
20     head[x] = cnt;
21 }
22
23 int son[maxn];
24 int top[maxn];
25 int dep[maxn];
26 int fa[maxn];
27 int size[maxn];
28
29 void dfs1(int u,int pre,int d){
30     dep[u] = d;
31     fa[u] = pre;
32     size[u] = 1;
33     int i;
34     for(i=head[u];i!=-1;i=E[i].next){
35         int v = E[i].v;
36         dfs1(v,u,d+1);
37         size[u] += size[v];
38         if(son[u] == -1 || size[v] > size[ son[u] ])
39             son[u] = v;
40     }
```

```
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
```

```

85     if( dep[x] > dep[y])
86         swap(x,y);
87     return x;
88 }
89
90 int main(){
91     memset(head,-1,sizeof(head));
92     memset(son,-1,sizeof(son));
93     scanf("%d%d",&m,&root);
94     int i;
95     for (i=1;i<=m;i++){
96         int x,y;
97         scanf("%d%d",&x,&y);
98         addedge(x,y);
99     }
100     dfs1(root,root,1);
101     dfs2(root,root);
102
103     scanf("%d",&n);
104     int x,y;
105     for(i=1;i<=n;i++){
106         scanf("%d%d",&x,&y);
107         int ans = find(x,y);
108         printf("%d\n",ans);
109     }
110
111     return 0;
112 }

```

练习题目 2

题目地址：[【模板】最近公共祖先（LCA）](#)

代码

```

1  #include<iostream>
2  #include<cstdio>
3  #include<cstring>
4  #define maxn 500005
5  using namespace std;
6  int m,n,root;
7  struct edge{

```

```
8     int next;
9     int v;
10 }E[maxn<<1]; //存两遍边
11 int head[maxn];
12 int son[maxn],top[maxn],size[maxn],fa[maxn],dep[maxn];
13 int cnt=0;
14 void addEdge(int x,int y){
15     cnt++;
16     E[cnt].v=y;
17     E[cnt].next=head[x];
18     head[x]=cnt;
19 }
20 void dfs1(int u,int pre,int d){
21     dep[u]=d;
22     fa[u]=pre;
23     size[u]=1;
24     for(int i=head[u];i!=-1;i=E[i].next){
25         int v=E[i].v;
26         if( v == pre ) continue;
27         dfs1(v,u,d+1);
28         size[u]+=size[v];
29         if(son[u]==-1 || size[v]>size[son[u]]){
30             son[u]=v;
31         }
32     }
33 }
34 void dfs2(int u,int sf){
35     top[u]=sf;
36     if(son[u]!=-1){
37         dfs2(son[u],top[u]);
38     }
39     else return ;
40     for(int i=head[u];i!=-1;i=E[i].next){
41         int v=E[i].v;
42         if(v!=son[u] && v != fa[u])
43             dfs2(v,v);
44     }
45 }
46 int find(int x,int y){
47     int f1=top[x],f2=top[y];
48     while(f1!=f2){
49         if(dep[f1]<dep[f2]){
50             swap(f1,f2);
51             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:

```

1 //暴力统计u结点,和轻儿子的信息
2 void dfs_count(int u){
3     dfs_count(v not hson[u])
4 }
5
6 //删除u结点为根的子树对 信息记录 的贡献
7 void del_count(int u){
8 }
9
10 //u: 当前结点,fa: 父结点,keep: 是否保留u的贡献
11 void dsu_on_tree(int u,int fa,bool keep){
12     // 1. 递归算轻儿子树
13     for(v in u.lighson){
14         dsu_on_tree(v,u,false);
15     }
16
17     // 2. 递归算重儿子树
18     if( son[u] ){
19         dsu_on_tree(son[u], u, true);
20         flag_hson = son[u];
21     }
22     // 3. 暴力统计u,和u的轻儿子子树的贡献
23     dfs_count(u,fa);
24     ans[u] = max_color_sum; //得到u结点子树的答案
25     if( !keep ){
26         // 从u回溯,删除子树上的所有点的贡献
27         //! 会把记录信息的数组清零
28         del_count(u, fa);
29         // 清空 必须放这里,想一想,只有重儿子的情况
30         max_color_sum = max_color_cnt = 0;
31     }
32 }
```

7 图论算法

7.1 最短路问题

7.1.1 Floyd-Warshall 算法

7.1.1.1 floyd

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

```

1 void floyd(){
2     int k,i,j;
3     for(k=1;k<=n;k++){ //k在最外层,枚举中间点
4         for (i=1;i<=n;i++)
5             for(j=1;j<=n;j++)
6                 if(f[i][k]+f[k][j]<f[i][j]){ //松弛法,如果能更小,那就更小
7                     f[i][j] = f[i][k]+f[k][j];
8                 }
9     }
10 }
```

7.1.1.2 floyd 最小环

```

1 for(k=1;k<=n;k++) {
2     for(i=1;i<k;i++) // 此时[i,j]之间的最短路还不经过k
3         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
4             if(f[i][j]+m[i][k]+m[k][j]<ans)
5                 ans=f[i][j]+m[i][k]+m[k][j];
6     for(i=1;i<=n;i++)
7         for(j=1;j<=n;j++)
8             if(f[i][k]+f[k][j]<f[i][j])
9                 f[i][j]=f[i][k]+f[k][j];
10 }
```

7.1.2 Dijkstra

```

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

```

6   while(!q.empty()){
7       int now=q.top().second;
8       q.pop();
9       if(vis[now])continue;
10      vis[now]=1;
11      for(int i=head[now];i!=-1;i=e[i].next){
12          if(dis[e[i].v]>dis[now]+e[i].w){
13              dis[e[i].v]=dis[now]+e[i].w;
14              q.push(make_pair(dis[e[i].v],e[i].v));
15          }
16      }
17  }
18 }

```

7.1.3 Bellman-ford

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

```

1 void bellman_ford(){
2     int i,j;
3     for(i=1;i<=n-1;i++)//进行n-1轮操作
4         for(j=1;j<=m;j++){
5             int &ss = u[j],&tt=v[j],&ww = w[j];//引用
6
7             if(dis[ss] > dis[tt] + ww)
8                 dis[ss] = dis[tt] + ww;
9
10            /* 如果是无向图 要反过求一次 */
11            if(dis[tt] > dis[ss] + ww)
12                dis[tt] = dis[ss] + ww;
13        }
14    //代码完成
15 }

```

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

```

1 void spfa(){
2     push(s);
3     dis[s] = 0;
4     inQueue[s] = 1;

```

```

5     pre[s] = -1;
6
7
8     int i;
9     while( empty() == false){
10         int t = pop(); //取队首
11         inQueue[t] = 0; // 不在队中
12
13         for(i = first[t]; i != -1; i = edge[i].next){
14             int tv = edge[i].v;
15             int tw = edge[i].w;
16             if( dis[tv] > dis[t] + tw){ //更新
17                 dis[tv] = dis[t] + tw;
18
19                 /* 不在队列中,就加入队列 */
20                 if( inQueue[tv] == 0){
21                     push(tv);
22                     inQueue[tv] = 1;
23                 }
24             }
25         }
26     }
27 }

```

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:

```

1  /* 求负环为什么dis清0? 为什么dis,ins只要清一次?
2  * */
3  namespace spfa_dfs{
4      using namespace xlx1;
5      bool ins[maxn]; //在栈内
6      int dis[maxn]; // double ,long long 根据题意更改
7      bool dfs(int u){ //找负环
8          ins[u] = 1;
9          for( int i = head[u]; ~i; i = e[i].next){
10              int v = e[i].v,w=e[i].w;
11              if( dis[v] > dis[u]+w){
12                  dis[v] = dis[u]+w;

```

```

13         if( ins[v] || dfs(v) ) return true;
14     }
15 }
16 ins[u] = 0;
17 return 0;
18 }
19 bool wk(){
20     memset(dis,0,sizeof(dis));
21     memset(ins,0,sizeof(ins)); // bool的全局变量可能不全是0
22     for(int i=1;i<=n;++i){
23         if( dfs(i) ) return 1;
24     }
25     return 0;
26 }
27 }

```

7.1.6 k 短路径

```

1  const int maxn = 1e5+5;
2  int dis[maxn];
3  struct node_for_astar_k {
4      int v,w;
5      friend bool operator<(const node_for_astar_k &a,const node_for_astar_k &b){
6          return a.w+dis[a.v] > b.w+dis[b.v];
7      }
8  };
9
10 int astart_Kshort_path(int s,int t){
11
12     /* 起点和终点相同是 */
13     if( s == t){
14         k++;
15     }
16     q.push(qnode(s,0));
17     while(!q.empty()){
18
19         qnode h = q.top();
20         q.pop();
21
22         int now = h.v;
23         if( now ==t){
24             cnt++;
25             if( cnt == k ) return h.w;

```

```

26         }
27
28         int i;
29         for(i = head[now]; i != -1; i = e[i].next){
30             int &v = e[i].v;
31             int &w = e[i].w;
32             q.push(qnode(v,w+h.w));
33         }
34
35     }
36
37     return -1;
38 }
39

```

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:

```

1  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
2  bool instack[maxn];
3  stack<int> sta;    // 栈
4  void tarjan(int u) {
5      dfn[u]=low[u]=++dfn;          // 为节点u设定次序编号和low初值
6      sta.push(u);                  // 将节点u压入栈中
7      in
8      for(int i = head[u]; ~i ; i = e[i].enxt){
9          int v = e[i].v;

```

```

10     if( !dfn[v]) {           // 如果节点v未被访问过
11         tarjan(v);           // 继续向下找
12         low[u] = min(low[u],low[v]);
13     }
14     else if( instack[v]){ // 反祖边,节点v还在栈内
15         low[u] = min(low[u],dfn[v]); //low[u] = min(low[u], low[v]) 理论上这样
16 写也可以
17     }
18
19 }
20 if( dfn[u] == low[u]){ // 如果节点u是强连通分量的根
21     color_cnt++;
22     int t = -1;
23     do {
24         t = sta.top(); sta.pop();
25         instack[t] = 0; // 将v退栈, 为该强连通分量中一个顶点
26         color[t] = color_cnt;
27     } while( u != v);
28 }
}

```

7.3 无向图连通性

7.3.1 图的割点

```

1  int root; //root点
2  void tarjan(int u){
3      int child=0;//记录root的孩子数
4      dfn[u] = low[u] = ++cnt; // 编号
5
6      int i;
7      for(i=head[u];i!=-1;i=E[i].next){ //遍历相邻点
8          int v = E[i].v ;//另一个点
9          if(! dfn[v] ){ //v点的编号为0, 也就是没有被访问
10             tarjan(v); //从这个点开始dfs
11             low[u] = min(low[u],low[v]); //树枝边
12             if( u == root) child++; // u是root点
13
14             if(low[v] >= dfn[u] && u != root) // 情况2
15                 cut[u] =1;
16         }
17         else //注意:v可能是u的父亲,但没有关系,最多low[u] == dfn[fa[u]]
18             low[u] = min(low[u],dfn[v]); //回边

```



```

19     }
20     // 退出这个点
21     if( u == root && child > 1) //情况1:
22         cut[u] = 1;
23 }

```

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 e 为 G 的桥，又称割边或关节边。

性质：对于一条边 $\langle u, v \rangle$ ， 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， $\langle u, v \rangle$ 是割边。 ($low[v] > dfn[u]$)

```

1 void CutEdge(int cur,int par)
2 {   dfn[cur]=low[cur]=++Index;
3
4     for(int i=head[cur];i!=-1;i=e[i].next)
5     {
6         int v=e[i].v;
7         if(v==par)continue; //注意这里和求割点的不同,这里不能用父亲跟新
8                               // 但求割点可以用父亲更新,想想为什么!
9         if(!dfn[v])
10        {
11            CutEdge(v,cur);
12            if(low[cur]>low[v])
13                low[cur]=low[v];
14            if(low[v]>dfn[cur])
15            {
16                ans[nAns++]=e[i].id;
17            }
18        }
19        else if(low[cur]>dfn[v])
20            low[cur]=dfn[v];
21    }
22 }

```

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

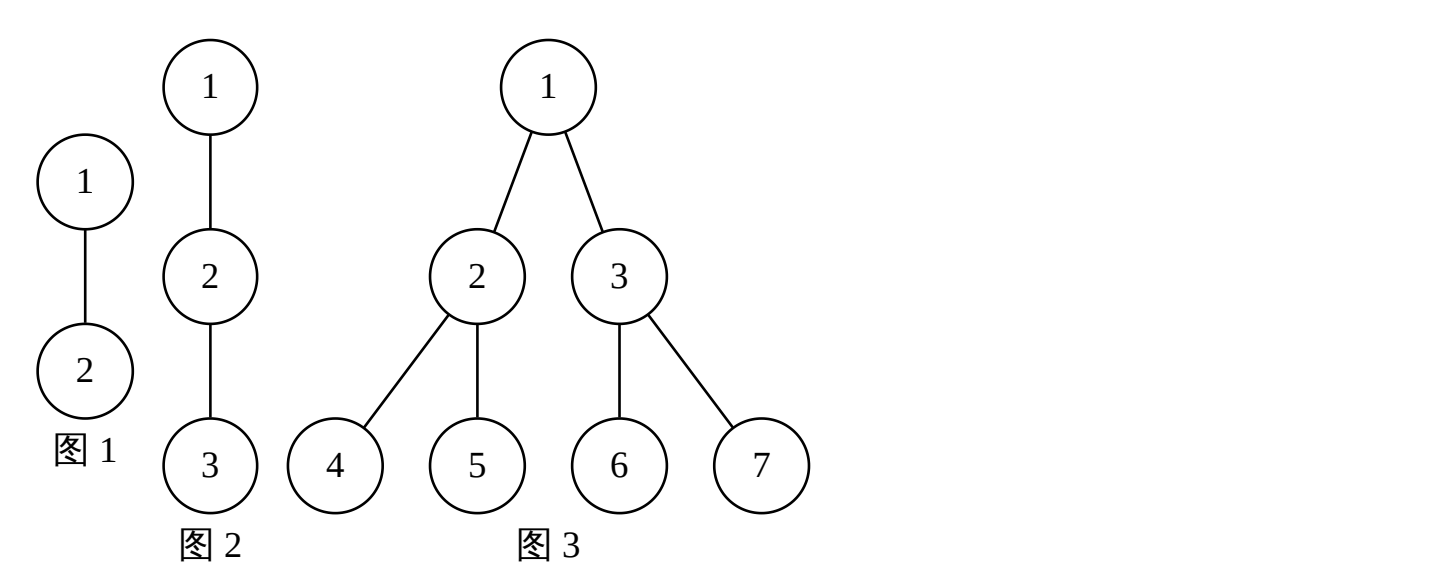
点双连通图的定义等价于任意两条边都同在一个简单环中

对于一个无向图，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



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 $\text{low}[v] \geq \text{dfn}[u]$ ，我们就将栈里的东西拿出来直到遇到当前边。

```

1  int dfn[maxn], low[maxn];
2  int idx, bcc_cnt;
3  stack<int> sta; //栈,存边的编号
4  //点的颜色,就是点属于哪个bcc
5  //防止bbc含有一个点多次
6  int color[maxn];
7  vector<int> bcc[maxn]; //属于某个bcc的点有哪些
8
9  void tarjan(int u){
10     dfn[u]=low[u] = ++idx;
11     int i;
12     for(int i=head[u]; ~i; i=e[i].next){
13         int v= e[i].v;
14         if( !dfn[v]){ //没有访问过
15             sta.push(i); //存边入栈
16             tarjan(v);
17             low[u] = min(low[u], low[v]);
18             //!!!!注意:
19             //这里没有判断u!=root
20             if( low[v] >= dfn[u]){
21                 bcc_cnt++;
22                 while(1){
23                     int i = sta.top(); sta.pop();
24                     int uu = e[i].u, vv = e[i].v;
25                     if(color[uu] != bcc_cnt){
26                         bcc[bcc_cnt].push_back(uu);
27                         color[uu] = bcc_cnt;
28                     }
29                     if(color[vv] != bcc_cnt){
30                         bcc[bcc_cnt].push_back(vv);
31                         color[vv] = bcc_cnt;
32                     }
33                     if( uu == u && vv == v) break;
34                 }
35             }
36         }
37     }
38 }

```

```

37         else low[u] = min(low[u],dfn[v]);
38     }
39
40 }
41 void find_bcc(){
42     int i;
43     for( i =1;i<=n;i++)
44         if( !dfn[i]) tarjan(i);
45 }
46

```

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

```

1  int low[maxn],dfn[maxn];
2  int col[maxm<<1];//用于标记桥
3  int ord=1;
4  void tarjan(int u,int pre){
5      low[u]=dfn[u]=ord++;
6      int i;
7      for(i=head[u];i!=-1;i=e[i].next){
8          int v=e[i].v;
9          if(v!=pre){
10             if(!dfn[v]){
11                 tarjan(v,u);
12                 low[u]=min(low[u],low[v]);
13
14                 if(low[v]>dfn[u]){
15                     col[i]=col[i^1]=1;//标记边及其反向边
16                     //注意: 边的编号从0开始
17                 }
18             }
19             else {
20                 low[u]=min(low[u],dfn[v]);
21             }
22         }
23     }
24 }
25

```

```

26 int mark;
27 int color[maxn]; //不同的边双连通分量会被染为不同的颜色
28 void dfs(int u, int pre){
29     color[u] = mark;
30
31     int i;
32     for(i = head[u]; i != -1; i = e[i].next){
33         int v = e[i].v;
34         if(!color[v] and v != pre and !color[v]){ //该边非桥, 不为前驱, 且没有被染色
35             dfs(v, u);
36         }
37     }
38 }
39 }
40
41 void handle(){
42     tarjan(1, 0); //原图连通
43
44     int i;
45     for(i = 1; i <= n; i++){
46         if(!color[i]){ //该点没有被染过
47             mark++;
48             dfs(i, 0);
49         }
50     }
51 }

```

7.3.4 割点，割边，点双，边双四合一模板

```

1  /*-----
2  * author: Rainboy
3  * email: rainboylvx@qq.com
4  * time: 2020年 05月 17日 星期日 18:46:05 CST
5  * problem: online_judge_id
6  *-----*/
7  #include <bits/stdc++.h>
8  #define For(i, start, end) for(i = start; i <= end; ++i)
9  #define Rof(i, start, end) for(i = start; i >= end; --i)
10 typedef long long ll;
11 using namespace std;
12
13 /* ===== 全局变量 ===== */
14 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);

```



```
147     }
148     printf("\n");
149 }
150
151 //=====
152 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

```
1  /* kahnp拓扑排序
2  *
3  * */
4  #include <stdio>
5  #include <cstring>
6
7  #define N 10000
8  int n,m;
9
10 int indgree[N] = {0}; //入度
11
12 int first[N];
13 int edge_cnt = 0;
14 struct _e{
15     int u,v,w,next;
16 }e[N];
17
18 void addEdge(int u,int v,int w){
19     edge_cnt++;
20     e[edge_cnt].u = u;
21     e[edge_cnt].v= v;
22     e[edge_cnt].w=w;
23     e[edge_cnt].next = first[u];
24     first[u] = edge_cnt;
25 }
26
27 /* 栈的操作 */
28 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
```

```

73 int main(){
74     memset(first,-1,sizeof(first));
75     scanf("%d%d",&n,&m);
76     int i,j,k;
77     int t1,t2;
78     for(i=1;i<=n;i++){
79         scanf("%d%d",&t1,&t2);
80         indgree[t2]++;
81         addEdge(t1,t2,0);
82     }
83     kahn();
84     return 0;
85 }

```

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

```

1  /* dfs拓扑排序
2   *
3   * */
4  #include <cstdio>
5  #include <cstring>
6
7  #define N 10000
8
9  int first[N];

```

```
10 int edge_cnt = 0;
11 struct _e{
12     int u,v,w,next;
13 }e[N];
14
15 void addEdge(int u,int v,int w){
16     edge_cnt++;
17     e[edge_cnt].u = u;
18     e[edge_cnt].v= v;
19     e[edge_cnt].w=w;
20     e[edge_cnt].next = first[u];
21     first[u] = edge_cnt;
22 }
23
24 int n,m;
25 bool instack[N] = {0};
26
27 int stack[N];
28 int stack_index = 0;
29
30 void push(int x){
31     stack[stack_index++] = x;
32 }
33
34
35 void topSort_dfs(int u){
36
37     int i;
38     for(i=first[u];i!=-1;i=e[i].next){
39         int v = e[i].v;
40         if( ! instack[v]){ //不在栈中,没有被输出
41             topSort_dfs(v);
42             //没有后趋
43         }
44     }
45     instack[u] = 1;
46     push(u);
47 }
48
49 int main(){
50     memset(first,-1,sizeof(first));
51     scanf("%d%d",&n,&m);
52
53     int i,j,k;
```

```

54     int t1,t2;
55     for (i=1;i<=m;i++){
56         scanf("%d%d",&t1,&t2);
57         addEdge(t1,t2,0);
58     }
59
60     for (i=1;i<=n;i++){
61         if( ! instack[i]) // 没有在栈中
62             topSort_dfs(i);
63     }
64
65     for(i=stack_index-1;i>=0;i--)
66         printf("%d ",stack[i]);
67     return 0;
68 }

```

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。

```

1  stack<int> sta,ans;
2  void Euler(int s){
3      sta.push(s); //起点入栈
4      while( !sta.empty() ){
5          int x = sta.top(),i = head[x];
6          //找到第一条未访问的边
7          while( i!=-1 && vis[i] ) i = e[i].next;
8          if( i!=-1){
9              sta.push(e[i].v);
10             head[x] = e[i].next;
11             //标记边,边从0开始编号
12             vis[i] = vis[i^1] = 1;
13         }
14         else { //退出这个点
15             sta.pop();
16             ans.push(x);
17         }
18     }
19 }

```

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 $O(n \log n)$ 的时间复杂度内预处理 f 数组:

```

1  for(i=1;i<=n;i++) f[i][0] = a[i]; //初始化
2
3  for(j=1;(1<<j)<=n;j++){ //1<<j 表示处理的范围
4      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
5  范围内
6      f[i][j] = max (f[i][j-1],f[i+(1<<(j-1))][j-1]);
    }

```

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

```

1  int query(int l,int r){
2      //第一种方法
3      int x = int( log(r-l+1)/log(2));
4
5      //第二种方法,这种写法比上面的写法慢
6      // 例如, luogu3865 就过不了
7      int k = 0;
8      while( (1<<(k+1)) <=(r-l+1) ) k++; // 最后2^{k+1} > r-l+1
9
10     //return max(f[l][k],f[r-(1<<k)+1][k]);
11     return max(f[l][x],f[r-(1<<x)+1][x]);
12 }

```

8.2 树状数组

8.2.1 树状数组：基础

```

1  #include <cstdio>
2  #include <cstring>
3
4  #define MAX 5000010
5  int n,m;
6  int c[MAX] = {0};
7

```

```
8 int lowbit(int x){
9     return x&(-x);
10 }
11
12 void update(int pos,int num){
13     while(pos<=n){
14         c[pos] += num;
15         pos +=lowbit(pos);
16     }
17 }
18
19 //sum(1,pos)
20 int query(int pos){
21     int sum = 0;
22     while(pos >0){
23         sum+=c[pos];
24         pos -= lowbit(pos);
25     }
26     return sum;
27 }
28
29 int main(){
30     scanf("%d%d",&n,&m);
31     int i,j,k;
32     for (i=1;i<=n;i++){
33         scanf("%d",&j);
34         update(i,j);
35     }
36
37     int t1,t2,t3;
38     for (i=1;i<=m;i++){
39         scanf("%d%d%d",&t1,&t2,&t3);
40
41         if( t1 == 1){
42             update(t2,t3);
43         }
44         else {
45             int ans = query(t3) - query(t2-1);
46             printf("%d\n",ans);
47         }
48     }
49     return 0;
50 }
```

8.2.2 逆序对

求逆序对代码:

```
1  #include <cstdio>
2
3  int n = 5;
4  int a[]={0,3,1,4,5,2};
5
6  int c[100]={0}; //存数状数组
7
8  int lowbit(int x){
9      return x & (-x);
10 }
11
12 void update(int pos,int num){
13     while(pos<=n){ //n代码数组A的长度
14         c[pos]+=num;
15         pos+=lowbit(pos);
16     }
17 }
18
19 int query(int pos){
20     int sum = 0;
21     while(pos > 0 ){
22         sum+=c[pos];
23         pos -= lowbit(pos);
24     }
25     return sum;
26 }
27
28
29 int main(){
30     int i;
31     for(i=1;i<=n;i++){
32         update(a[i],1);
33         printf("%d ",i-query(a[i])); // 输出前面有几个数比自己大
34     }
35     return 0;
36 }
```

8.2.3 区间修改，单点查询


```

1  const int MAXN = 10000;//最多的点
2  int a[MAXN];// 原数组
3  int c[MAXN] = {0};// 树状数组
4
5  //原单点更新
6  void update(int pos,int num){
7      while(pos<=MAXN){
8          c[pos]+=num;
9          pos+=lowbit(pos);
10     }
11 }
12
13 //修改一段区间的值
14 void update_range(int i,int j,int n){
15     update(i,n);
16     update(j+1,-n);
17 }
18 //初始化,形成数状数组
19 for(i=1;i<=MAXN;i++){
20     update(i,a[i]- a[i-1])
21 }
22
23 //查询,也就是单点的值
24 int query(int pos){
25     int sum = 0;
26     while( pos > 0 ){
27         sum += c[pos];
28         pos -= lowbit(pos);
29     }
30     return sum;
31 }

```

8.2.4 区间增减 区间查询

```

1  void update(int pos,int num){
2      int t = pos;
3      while(pos<=n){ //n代码数组A的长度
4          c1[pos]+=num;
5          c2[pos] += t*num;
6          pos+=lowbit(pos);
7      }
8  }
9  void update_range(int i,int j,int n){

```

```

10     update(i,n);
11     update(j+1,-n);
12 }
13
14 int query1(int pos){
15     int sum=0;
16     while(pos >0){
17         sum += c1[pos];
18         pos -= lowbit(pos);
19     }
20     return sum;
21 }
22
23 int query2(int pos){
24     int sum=0;
25     while(pos >0){
26         sum += c2[pos];
27         pos -= lowbit(pos);
28     }
29     return sum;
30 }
31
32 int sum(i,j){
33     return (j+1)*query1(j) - query2(j) -i*query1(i-1) +query2(i-1);
34 }

```

8.2.5 树状数组 区间最值

```

1  /* pos 位置,v 数值 */
2  void update(int pos,int v){
3      int i,lb;
4      c[pos] = a[pos] = v;
5      lb = lowbit(pos);
6      for(i=1;i<lb;i <=1){ //利用孩子更新自己
7          c[pos] = c[pos] > c[pos-i] ? c[pos] : c[pos-i];
8      }
9      int pre = c[pos];
10     pos+=lowbit(pos);//父亲的位置
11
12     /* 更新父亲 */
13     while(pos <= n){
14         if( c[pos] < pre){ //更新的父亲
15             c[pos] = pre;

```

```

16         pos += lowbit(pos);
17     }          //没有更新父亲
18     else break;
19 }
20 }

```

8.2.6 二维树状数组

```

1 void update(int x,int y,int val){
2     a[x][y] += val;
3     int i,j;
4     for(i= x; i <= 横高度; i += lowbit(i) )
5         for( j = y ;j <= 纵宽度 ;j += lowbit(j))
6             c[i][j] += val;
7 }

```

```

1 int query(int x,int y){
2     int i,j,sum = 0;
3     for(i = x ; i >0 ; i -= lowbit(i) )
4         for(j=y;j>0;j-=lowbit(j))
5             sum += c[i][j];
6     return sum;
7 }

```

8.2.7 综合模板

```

1  /* ===== 树状数组 BIT
2  * 1.单点增减, 区间和
3  *     1.1 逆序对
4  * 2.区间增减, 单点值
5  * 3.区间增减, 区间和
6  * 4.单点修改, 末尾压入, 区间最值
7  * */
8  namespace bit {
9      typedef long long ll;
10     ll c[maxn],SIZE=maxn;
11     ll c2[maxn]; // c2[i] = i*c[i]
12     inline void bit_init(){}
13
14     /* 区间和 */
15     inline ll lowbit(ll x) { return x & -x;}
16     void update(ll pos,ll add){ while(pos<=SIZE) c[pos]+=add,pos+=lowbit(pos);}

```

```

17  /* 差分,区间增减,单点查询*/
18  void update_range(ll l,ll r,ll add){ update(l,add);update(r+1,-add); }
19  ll query(int pos){ll sum=0;while(pos>0) sum+=c[pos],pos-=lowbit(pos); return
20  sum;}
21
22  /* 差分,区间增减,区间查询*/
23  //核心公式: sum_a[i] = (i+1)*sum_c[i] - sum_{i*c[i]}
24  void update_c_c2(ll pos,ll add){ //同时更新c, c2
25      ll t = pos; while( pos <= SIZE){ c[pos] += add; c2[pos] += t*add;
26  pos+=lowbit(pos);}
27  }
28  void update_range_c_c2(ll l,ll r ,ll add){ update_c_c2(l,
29  add);update_c_c2(r+1, -add); }
30  ll query2(ll pos){ ll sum = 0; while( pos > 0) sum += c2[pos], pos -
31  =lowbit(pos); return sum; }
32  ll query_range_sum(ll l,ll r){ return (r+1)*query(r) - query2(r) - l*query(l-
33  1) + query2(l-1); }
34
35  // ===== 4.单点修改, 末尾压入, 区间最值
36  ll a[maxn]; // 原数组
37  void update_by_child(ll pos,ll v){ // alias push
38      c[pos] = a[pos] = v;
39      ll i,lb = lowbit(pos);
40      for(i=1 ; i < lb ; i <= 1) c[pos] = std::max(c[pos],c[pos-i]);
41  }
42
43  void update_ex(ll pos,ll v){
44      update_by_child(pos,v); int tmp = c[pos];
45      for( pos += lowbit(pos); pos <=SIZE; pos+=lowbit(pos)){
46          if( c[pos] < tmp) c[pos] = tmp;
47          else break; //没有更新父亲
48      }
49  }
50  ll query_ex(ll l ,ll r){
51      ll ex = -1;
52      while( l <= r){
53          ll left = r - lowbit(r) +1; //范围内的最左点
54          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

```

1 namespace BCJ {
2     int fa[maxn];
3     inline void bcj_init(int x){ for(int i=1;i<=x;i++) fa[i] = i; }
4     int find(int x){ if( x == fa[x]) return x; return fa[x] = find(fa[x]); }
5     void merge(int x,int y){ fa[find[x]] = find(y); }
6 }
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

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)

```

1 #include <cstdio>
2
3 #define lson(rt) (rt<<1)
4 #define rson(rt) (rt<<1)|1
5 #define maxn 55555
6 int tree[maxn<<2];
7
8 void pushup(int rt){
9     /* 不同的题目有不同的写法 */
10    tree[rt] = tree[lson(rt)] + tree[rson(rt)];
11 }
12
13 void build(int l,int r,int rt){
14     if(l == r){
15         scanf("%d",tree[rt]);//想一想,为什么这样可以读取呢?
16         return;
17     }
18     int m = (l+r)>>1;
19     build(l,m,lson(rt));
20     build(m,r,rson(rt));
21     pushup(rt);
22 }
23
24 int main(){
25     int n;
26     while(scanf("%d",&n)!=EOF){
27         build(1,n,1);
28         int q;
29         while(scanf("%d",&q)!=EOF){
30             int x,y;
31             if(q==1){
32                 scanf("%d",&x);
33                 tree[x]++;
34                 pushup(x>>1);
35             }else if(q==2){
36                 scanf("%d",&x);
37                 tree[x]--;
38                 pushup(x>>1);
39             }else if(q==3){
39                 scanf("%d",&x);
40                 printf("%d\n",tree[x]);
41             }
42         }
43     }
44     return 0;
45 }

```

```

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pos */
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,rson(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);

```

```

61         if(op[0] == 'Q')
62             printf("%d\n",query(a,b,1,n,1));
63         else if(op[0] == 'S')
64             update(a,-b,1,n,1);
65         else
66             update(a,b,1,n,1);
67     }
68 }
69 }

```

8.4.2 成段更新

```

1  /*=====
2  * Title : 线段树 成段替换
3  * Author: Rainboy
4  * Time  : 2016-05-27 13:05
5  * update: 2016-05-27 13:05
6  * ? Copyright 2016 Rainboy. All Rights Reserved.
7  *=====*/
8
9  #include <stdio>
10 #include <string>
11 const int maxn = 1000;
12 int st[maxn<<2];
13 int flag[maxn<<2];
14 #define lson(rt) (rt<<1)
15 #define rson(rt) ((rt<<1)|1)
16 int n,m;
17 void pushup(int rt){
18     st[rt] = st[lson(rt)] + st[rson(rt)];
19 }
20 void pushdown(int rt,int m){
21     if(flag[rt]){
22         flag[lson(rt)] = flag[rson(rt)] = flag[rt];
23         st[lson(rt)] = flag[rt]*(m-(m>>1)); st[rson(rt)] = flag[rt]*(m>>1);
24         flag[rt] = 0;
25     }
26 }
27 void update(int l1,int r1,int c,int l,int r,int rt){
28     if(l1 <=l && r<=r1){
29         flag[rt] = c; //我们到达一个点
30         st[rt] = (r-l+1)*c;
31         return ;

```

```
32     }
33     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
34     int m = (l+r)>>1;
35     if( l1 <= m) update(l1,r1,c,l,m,lson(rt));
36     if( r1 > m) update(l1,r1,c,m+1,r,rson(rt));
37     pushup(rt);
38 }
39 int query(int l1,int r1,int l,int r,int rt){
40     if(l1<=l && r <= r1){//包含
41         return st[rt];
42     }
43
44     //路过
45     pushdown(rt,(r-l+1));
46     int ret = 0;
47     int m = (l+r)>>1;
48     if(l1 <= m) ret+= query(l1,r1,l,m,lson(rt));
49     if(r1 > m ) ret+= query(l1,r1,m+1,r,rson(rt));
50     return ret;
51 }
52 void build(int l,int r,int rt){
53     if( l==r){
54         scanf("%d",&st[rt]);
55         return ;
56     }
57     int m = (l+r)>>1;
58     build(l,m,lson(rt));
59     build(m+1,r,rson(rt));
60     pushup(rt);
61 }
62 int main(){
63     memset(flag,0,sizeof(flag));
64     scanf("%d",&n);
65     build(1,n,1);
66     scanf("%d",&m);
67     int i,j,k;
68     char c;
69     while(m--){
70         scanf("%c",&c); //读两次,滤掉\n
71         scanf("%c",&c);
72         if( c == 'c' ){
73             scanf("%d%d%d",&i,&j,&k);
74             update(i,j,k,1,n,1);
75         }else {
```



```

76         scanf("%d%d",&i,&j);
77         int ans = query(i,j,1,n,1);
78         printf("%d\n",ans);
79     }
80 }
81 return 0;
82 }
83

```

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

```

1  int merge(int a,int b,int l,int r){
2      if(!a) return b;
3      if(!b) return a;
4      if( l == r){
5          // 按题目意思 合并
6          // tr[a].val += tr[b].val
7          return a;
8      }
9
10     int md = (l+r) >> 1;
11     tr[a].l = merge(tr[a].l,tr[b].l,l,md);
12     tr[a].r = merge(tr[a].r,tr[b].r,md+1,r);
13     push_up(a);
14     return a;
15 }

```

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

```

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

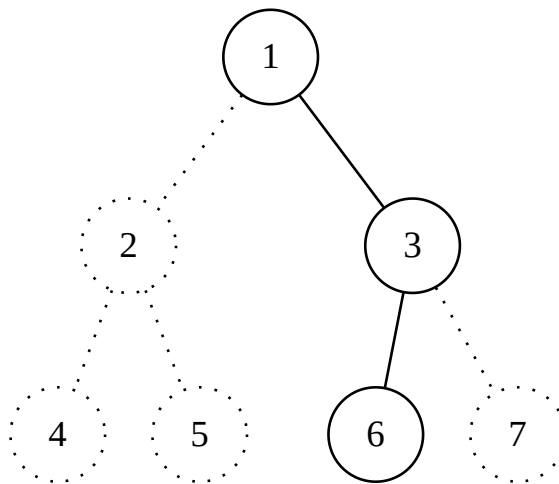
```

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 $lson = rt \ll 1, rson = (rt \ll 1) | 1;$

```
1 struct node {
2     int val,l,r; //值,左,右孩子的下标
3 }
```

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

```
1 struct { int val,l,r,flag;} tree[maxn];
2 int cnt ,root;
3 void newnode (int &p,int val){
4     p = ++cnt;
5     tree[p].val= val;
6     tree[p].l= tree[p].r = 0;
7 }
```

```
1 void push_up(int rt){
2 }
3 void push_down(int rt){
4 }
```

```

1 void update(int u,int L,int R,int x,int val){
2 }
3
4 void update_range(int u,int L,int R,int x)
5 int query(int u,int L,int R,int l,int r){
6 }

```

8.6 主席树

8.6.1 主席树：入门

luogu3834

```

1  /*-----
2  * author: Rainboy
3  * email: rainboylvx@qq.com
4  * time: 2019年 10月 14日 星期一 17:53:55 CST
5  * problem: luogu-3834
6  *-----*/
7  #include <bits/stdc++.h>
8  using namespace std;
9
10 const int maxn = 2e5+5; int n,m;
11
12 int a[maxn]; //原数组
13 int b[maxn]; //原数组
14
15 int root[maxn];
16 struct Node {
17     int l,r,sum;
18 };
19
20 int cnt=0;
21 Node tree_node[maxn*40];
22 inline int get_tree_node(){
23     return ++cnt;
24 }
25
26 namespace LSH { //离散化
27     int idx=1,i;
28     //使用一个数组进行离散化
29     int lsh(int arr[],int len){
30         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;
```

```

75     for(i=1;i<=n;i++){
76         scanf("%d",&a[i]);
77         b[i] = a[i];
78     }
79     sort(b+1,b+n+1);
80     LSH::lsh(b,n); //离散化
81 }
82
83
84 int main(){
85     init();
86     /* 建立树 */
87     int i;
88     for(i=1;i<=n;i++){
89         insert(1, n, root[i-1], root[i], LSH::get_id(b,a[i]) );
90     }
91     for(i=1;i<=m;i++){
92         int l,r,k;
93         scanf("%d%d%d",&l,&r,&k);
94         int ans = query(1,n,root[l-1],root[r],k);
95         printf("%d\n",b[ans]);
96     }
97     return 0;
98 }
99

```

8.7 平衡树

8.7.1 替罪羊树 [Scapegoat Tree]

```

1  const double alpha  = 0.75;
2  const double del_alpha = 0.3;
3  const int base_node = 5;
4
5  /* 是否平衡 */
6  bool imbalance(int now){
7      if( max(tzy[tzy[now].l].size , tzy[tzy[now].r].size) > tzy[now].size*alpha +
8  base_node
9      || tzy[now].size - tzy[now].fact > tzy[now].size*del_alpha + base_node)
10         return true;
11         return false;
12 }
13

```

```
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 */
```

```

58 void update(int now,int end){
59     if(!now) return;
60     if( tzy[end].val < tzy[now].val){
61         update(tzy[now].l, end);
62     }
63     else update(tzy[now].r, end);
64     tzy[now].size = tzy[ tzy[now].l ].size + tzy[ tzy[now].r ].size+1;
65 }
66
67 void check(int &now,int end){
68     if( now == end) return;
69     if( imbalance(now)){
70         rebuild(now);
71         update(root, now);
72         return;
73     }
74     if( tzy[end].val < tzy[now].val)
75         check(tzy[now].l, end);
76     else check(tzy[now].r, end);
77 }

```

8.7.2 Splay 入门

```

1  /*-----
2  * author: Rainboy
3  * email: rainboylvx@qq.com
4  * time: 2019年 11月 17日 星期日 15:26:49 CST problem: luogu-3369
5  *-----*/
6  #include <bits/stdc++.h>
7  using namespace std;
8
9  /* ===== 全局变量 =====*/
10 const int maxn = 1e5+5;
11 int n;
12 /* ===== 全局变量 END =====*/
13
14 /* ===== 快读 ===== */
15 void in(int &a){
16     a = 0;
17     int flag = 1;
18     char ch = getchar();
19     while( ch < '0' || ch > '9'){ if( ch == '-' ) flag = -1; ch = getchar(); }
20     while( ch >= '0' && ch <= '9'){ a = a*10 + ch-'0'; ch = getchar(); }

```

```

21     a = a*flag;
22 }
23 /* ===== 快读 END ===== */
24
25 struct Node
26 {
27     int fa,ch[2],val,cnt,size; //ch[0]是左儿子, ch[1]是右儿子
28 }spl[maxn];
29 int cnt,root; //内存池计数,根编号
30
31 //新建节点, 要注意fa指针的初始化
32 void newnode(int &now,int fa,int val)
33 {
34     spl[now=++cnt].val=val;
35     spl[now].fa=fa;
36     spl[now].size=spl[now].cnt=1;
37 }
38
39 bool ident(int x,int f){ return spl[f].ch[1] == x; }
40
41 void connect(int x,int fa,int ch){
42     spl[x].fa = fa;
43     spl[fa].ch[ch] = x;
44 }
45
46 inline void push_up(int x){
47     spl[x].size=spl[spl[x].ch[0]].size+spl[spl[x].ch[1]].size+spl[x].cnt;
48 }
49
50 void rotate(int x) //合二为一的旋转
51 {
52     // f:父亲,ff:祖父,k:x是父亲的那个孩子
53     int f=spl[x].fa,ff=spl[f].fa,k=ident(x,f);
54     connect(spl[x].ch[k^1],f,k); //x的孩子作为父亲f的孩子
55     connect(x,ff,ident(f,ff)); //x作为祖父的孩子
56     connect(f,x,k^1); //f作为x的孩子
57     push_up(f),push_up(x); //别忘了更新大小信息
58 }
59
60 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
61 {
62     // !top => top == 0
63     if(!top)
64         root=x; //改变根结点

```



```

65 while(spl[x].fa!=top) //
66 {
67     int f=spl[x].fa,ff=spl[f].fa;
68     //祖父不是目的结点,要旋转两次,因为有三个点
69     if(ff!=top)
70         rotate( ident(f,ff)^ident(x,f)? x : f );
71     //直线:旋转父亲
72     //之字:旋转自己
73     rotate(x);    //最后一次都是旋转自己
74 }
75 }
76
77 // fa默认为0,now默认为root
78 void ins(int val,int &now=root,int fa=0)    //递归式, 要传fa指针
79 {
80     if(!now) //当前是空结点,建立,伸展到root
81         newnode(now,fa,val),splaying(now,0);
82     else if(val<spl[now].val) //去左子树
83         ins(val,spl[now].ch[0],now);
84     else if(val>spl[now].val) //去右子树
85         ins(val,spl[now].ch[1],now);
86     else //相等,计数+1,伸展到root
87         spl[now].cnt++,splaying(now,0);
88 }
89
90 void delnode(int x)
91 {
92     splaying(x,0);//把x伸展到root
93     if(spl[x].cnt>1) spl[x].cnt--; //计数-1
94     else if(spl[x].ch[1]) //存在右子树
95     {
96         int p = spl[x].ch[1]; //p 右子树编号
97         //p 右子树中的最小值
98         while(spl[p].ch[0]) p=spl[p].ch[0];
99         splaying(p,x); //p 伸展到 x的孩子
100
101         //x的左孩子,变p的左孩子
102         connect(spl[x].ch[0],p,0);
103         root=p; //根变p
104         spl[p].fa=0; //根的父亲
105         push_up(root); //更新size
106     } // 只存在左子树,直接做
107     else root=spl[x].ch[0],spl[root].fa=0;
108 }

```

```
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     //=====
```

```

197     fprintf(stderr, "\n Total Time: %lf ms", double(clock()-
198 program_start_clock)/(CLOCKS_PER_SEC / 1000));
199     return 0;
200 }

```

8.7.3 Treap

```

1  /* Author:Rainboy 2018-09-08 00:32 */
2  #include <cstdio>
3  #include <cstring>
4
5  #define N 100005
6  #define ls tr[p].l    //左孩子
7  #define rs tr[p].r    //右孩子
8  const int INF = 0x7fffffff;
9
10 int n;
11
12 struct node{
13     int l,r,val; //左右孩子,点的值
14     int size,rand,cnt; //子树的大小,随机值,该结点出现的次数
15 } tr[N];
16 int sz = 0; //编号用
17
18 int rmax(int a,int b){
19     if(a > b ) return a;
20     return b;
21 }
22 int rmin(int a,int b){
23     if(a < b ) return a;
24     return b;
25 }
26
27
28
29 inline int rand ( ) {
30     static int seed = 733;
31     return seed = ( int ) seed * 482711LL % 2147483647;
32 }
33
34 /* 更新当前点的size */
35 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 {

```

```

80     if (p==0) return;
81     if (tr[p].val==x)
82     {
83         if (tr[p].cnt>1) tr[p].cnt--,tr[p].size--;//如果有多个直接减一即可。
84         else
85         {
86             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。
87             else if (tr[ls].rand<tr[rs].rand) rturn(p),del(p,x);
88             else lturn(p),del(p,x);
89         }
90     }
91     else if (x>tr[p].val) tr[p].size--,del(rs,x);
92     else tr[p].size--,del(ls,x);
93 }
94
95
96 /* 找到排名 ,所有比x点小的点有多少个*/
97 int find_pm(int p,int x){
98     if(p==0) return 0;
99     if(tr[p].val == x) return tr[ls].size+1;
100    if(x > tr[p].val ) //x比当前点大,进入右子树
101        return tr[ls].size+tr[p].cnt+find_pm(rs,x);
102    else //x比当前点要小于
103        return find_pm(ls,x);
104 }
105
106 /* 查询排名为x的数 */
107 int find_sz(int p,int x){
108     if(p ==0 ) return 0;
109     if( x <=tr[ls].size)
110         return find_sz(ls,x);
111
112     x -= tr[ls].size;
113     if(x<=tr[p].cnt ) return tr[p].val;
114     x -= tr[p].cnt;
115     return find_sz(rs,x);
116 }
117
118 /* 找到前趋 */
119 int find_qq(int p,int x){
120     if(p == 0 ) return -INF;
121     if( tr[p].val <x )
122         return rmax(tr[p].val,find_qq(rs,x));
123     else

```

```

124         return find_qq(ls,x);
125     }
126     /* 找到后继 */
127     int find_hj(int p,int x){
128         if(p == 0 ) return INF;
129         if( tr[p].val <= x)
130             return find_hj(rs,x);
131         else
132             return rmin(tr[p].val,find_hj(ls,x));
133     }
134
135     int main(){
136         scanf("%d",&n);
137
138         int i,flag,x,rt=0;
139         for (i=1;i<=n;i++){
140             scanf("%d%d",&flag,&x);
141             if( flag == 1)
142                 insert(rt,x);
143             else if( flag == 2)
144                 del(rt,x);
145             else if( flag == 3)
146                 printf("%d\n",find_pm(rt,x));
147             else if( flag == 4)
148                 printf("%d\n",find_sz(rt,x));
149             else if( flag == 5)
150                 printf("%d\n",find_qq(rt,x));
151             else if( flag == 6)
152                 printf("%d\n",find_hj(rt,x));
153         }
154         return 0;
155     }

```

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 树中的值都大于 val

root 为当前 dfs 树中子树的根

- 如果 root 点的 val \leq 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

```

1 void split(int now,int val,int &x,int &y){
2     if( !now ) {
3         x = y = 0;
4         return;
5     }
6
7     if( fhq[now].val <= val){
8         x= now;
9         split(fhq[now].r, val, fhq[now].r, y);
10    }
11    else { // >
12        y = now;
13        split(fhq[now].l, val, x, fhq[now].l);
14    }
15    update(now);
16 }

```

8.8 link cut tree(LCT)

8.8.1 「LCT 入门」序

```

1 const int maxn = 1e5+5;
2 struct Node
3 {
4     int fa,ch[2],val,res;    //res是异或结果
5     bool flag;              //翻转懒标记
6 }spl[maxn];
7 //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
8 #define ls(x) (spl[x].ch[0])
9 #define rs(x) (spl[x].ch[1])
10 #define fa(x) (spl[x].fa)
11 #define ident(x,f) (rs(f)==x)    //和下面的connect都是Splay的辅助函数
12 #define connect(x,f,s) spl[fa(x)=f].ch[s]=x
13 #define update(x) spl[x].res=spl[ls(x)].res^spl[rs(x)].res^spl[x].val
14 #define ntroot(x) (ls(fa(x))==x||rs(fa(x))==x) //判断结点是否为Splay的根
15 #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

```

1 int read(){
2     int x=0,t=1;
3     char ch=getchar();
4     while(ch<'0' || ch>'9'){ if(ch=='-') t=-1; ch=getchar(); }
5     while(ch<='9' && ch>='0') { x=x*10+ch-'0'; ch=getchar(); }
6     return x*t;
7 }

```

更快的快读与快输 (补充 by Rainboy)

```

1 #include <iostream>
2 #include <ctime>
3 #include <cstdio>
4 #include <cctype>
5 namespace FastIO          //使用命名空间
6 {
7     char buf[1 << 21], buf2[1 << 21], a[20], *p1 = buf, *p2 = buf, hh = '\n';
8     int p, p3 = -1;
9     inline int getc()
10    {
11        // *p1++
12        return p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, 1 << 21, stdin), p1 ==
13 p2) ? EOF : *p1++;
14    }
15    inline int read()
16    {
17        int ret = 0, f = 0;
18        char ch = getc();
19        while (!isdigit(ch))
20        {
21            if (ch == '-')
22                f = 1;
23            ch = getc();
24        }
25        while (isdigit(ch))
26        {

```

```

27         ret = ret * 10 + ch - 48;
28         ch = getc();
29     }
30     return f ? -ret : ret;
31 }
32 inline void flush()
33 {
34     fwrite(buf2, 1, p3 + 1, stdout), p3 = -1;
35 }
36 inline void print(int x)
37 {
38     if (p3 > 1 << 20)
39         flush();
40     if (x < 0)
41         buf2[++p3] = 45, x = -x;
42     do
43     {
44         a[++p] = x % 10 + 48;
45     } while (x /= 10);
46     do
47     {
48         buf2[++p3] = a[p];
49     } while (--p);
50     buf2[++p3] = hh;
51 }
52 }
53 #define read() FastIO::read()
54 #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

```

1     sort(a+1, a+1+n);
2
3     int l=1,r=n;
4     int ans = 0;
5
6     while( l < r){
7         if(a[l] + a[r] <= k){

```

```
8         ans += r - l;  
9         l++;  
10    }  
11    else  
12        r--;  
13 }
```

9.3 离散化

```
1  #include <bits/stdc++.h>  
2  using namespace std;  
3  
4  int a[100];  
5  int b[100];  
6  int n;  
7  
8  int main(){  
9      int i,j;  
10     for (i=1;i<=n;i++){  
11         scanf("%d",&a[i]);  
12         b[i] = a[i];  
13     }  
14     sort(b+1,b+n+1);  
15     int nn = unique(b+1,b+n+1) - b - 1;  
16     for(i = 1;i<=n;i++){  
17         a[i] = lower_bound(b+1,b+n+1,a[i]) - b;  
18     }  
19     return 0;  
20 }
```