

算法-快速幂

ZeitHaum

2023 年 4 月 10 日

目录

分治算法的一种变形。带测试的递归标准写法 (USACO):

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int quickpow(int x,int y){
5     if(y==0) return 1;
6     int temp = quickpow(x,y/2);
7     if(y%2==1) return x*temp*temp;
8     else return temp*temp;
9 }
10
11 int main(){
12     cout<<quickpow(2,7)<<endl;
13 }
```

带测试的非递归标准写法 (USACO):

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int quickpow(int x,int y){
5     int ret = 1;
6     while(y>0){
7         //pow(x,y) = pow(x^2,y/2);
8         if(y&1) ret*=x;
9         x = x*x;
10        y = y>>1;
11    }
12    return ret;
13 }
14
15 int main(){
16     cout<<quickpow(2,7)<<endl;
17 }
```

算法-搜索

ZeitHaum

2023 年 6 月 28 日

目录

1 二分搜索	1
1.1 通用模板	1
1.2 C++ 相关库函数	2
1.2.1 lower_bound	2
1.2.2 upper_bound	2
1.2.3 binary_search	2
1.3 相关题目解析	2
1.3.1 力扣 287-寻找重复数	2
1.3.2 力扣 300-最长上升子序列 (LIS) 问题	3
1.4 概念区别	4
2 反对角线搜索	4
2.1 一般概述	4
2.2 复杂度分析	5
2.3 例题	5
2.3.1 力扣 11	5
参考资料	5

1 二分搜索

Stop leaning useless algorithm, go and solve some problems, learn how to use binary search.——Um_nik

1.1 通用模板

标准的二分模板至关重要。可以二分查找的数组需要满足：

1. 数组前面的元素都符合性质 1。
2. 数组后面的元素都符合性质 2。
3. 性质 1 和性质 2 是互斥对立的，一个元素要么属于性质 1，要么属于性质 2。

定义 l 为符合性质 1 的最右侧元素的索引， r 为符合性质 2 的最左侧元素的索引。显然有 $r - l = 1$ 。

编程时需要将 l 赋值为左边第一个满足性质 1 的索引， r 为右边最后一个满足性质 2 的索引。如果不存在则允许越界。即默认向数组最左侧和最右侧分别添加一个满足性质 1 和性质 2 的元素。

此时定义中间值 $mid = l + (r - l) / 2$ ，此时便不会出现 $mid = l$ 或 $mid = r$ 无法停止循环的问题。典型模板如下：

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int data[5] {1,3,5,6,8};
5
6 auto binary_search(int n,int target){
7     //二分查找数组第一个大于等于target的位置。
8     //性质1: 小于target.
9     //性质2: 大于等于target.
10    int l = -1;
11    int r = n;
12    while(r - l > 1){
13        int mid = l + (r - l) / 2;
14        auto check = [&]() {
15            if(data[mid] < target) return 1; //满足性质1
```

```
16         else return 2;
17     };
18     if(check()==1) l = mid;
19     else r = mid;
20 }
21 return r;
22 }
23
24 int main(){
25     auto p = binary_search(5,6);
26     cout<<p;//data[3] = 6,输出3.
27 }
```

1.2 C++ 相关库函数

1.2.1 lower_bound

返回第一个大于等于 target 的元素索引 (迭代器)。参数列表:

```
1 lower_bound(first,end,target)->iterator;
```

1.2.2 upper_bound

返回第一个大于 target 的元素索引 (迭代器)。参数列表:

```
1 upper_bound(first,end,target)->iterator;
```

1.2.3 binary_search

返回元素是否在指定范围中。参数列表:

```
1 binary_search(first,end,target)->bool;
```

1.3 相关题目解析

1.3.1 力扣 287-寻找重复数

链接:[寻找重复数](#)。

此题较为巧妙，因为二分的集合不是给定数组而是给定范围区间。对于处于 $[1, n]$ 的整数 i ，记函数 $f(i)$ 为数组中小于等于 i 的元素个数，可以发现 $f(i)$ 满足二分性。复杂度 $\Theta(n \log(n))$ 。

代码:

```
1 class Solution {
2 public:
3     int findDuplicate(vector<int>& nums) {
4         int n = nums.size()-1;
5         int l = 0;
6         int r = n;
7         while(r-l>1){
8             int mid = l + (r - l)/2;
9             auto check = [&]() {
10                 int cnt = 0;
11                 for(int i = 0; i<n+1; i++){
12                     if(nums[i]<=mid) cnt++;
13                 }
14                 if(cnt<=mid) return 1;
15                 else return 2;
16             };
17             if(check()==1) l = mid;
18             else r = mid;
19         }
20         return r;
21     }
22 };
```

1.3.2 力扣 300-最长上升子序列 (LIS) 问题

典型做法是 $\Theta(n^2)$ 的 DP，根据 Dilworth 定理可以提出 $\Theta(n \log(n))$ 的算法。Dilworth 定理的可推导出：一个数组的最长严格上升子序列长度等于最短的非严格下降子序列的划分。于是可以通过简单模拟非严格下降子序列的划分做法（贪心 + 二分）求得。

代码:

```
1 class Solution {
2 public:
```



```

3   int lengthOfLIS(vector<int>& nums) {
4       int n = nums.size();
5       vector<int> dp(n, 0);
6       dp[0] = 1;
7       int ans = 1;
8       for(int i = 1; i < n; i++){
9           int maxlen = 0;
10          for(int j = 0; j < i; j++){
11              if(nums[j] < nums[i]) maxlen = max(maxlen, dp[j]);
12          }
13          dp[i] = maxlen + 1;
14          ans = max(dp[i], ans);
15      }
16      return ans;
17  }
18 };

```

1.4 概念区别

严格递增是不包含相等的递增。非严格递增等价于非递减，包含相等的情况。

相反的概念类似。

2 反对角线搜索

2.1 一般概述

对于二元函数 $f(x, y)$ ，假设其值域为 $[1, n] \times [1, m]$ ($x, y, n, m \in \mathbb{R}$)。

设 (i, j) 为当前搜索值，

若存在一个关于 (i, j) 的判定函数 $Q(i, j)$ 使得：当 $Q(i, j) = 1$ 时，对于任意 $x < i$ ，均有 $f(i, j)$ 优于 $f(x, j)$ ；当 $Q(i, j) = 2$ 时，对于任意 $y > j$ ，均有 $f(i, j)$ 优于 $f(i, y)$ ；

则可以使用反对角线进行搜索。

定义“优于”：对于 (i_1, j_1) 和 (i_2, j_2) ，若二者都属于解空间，则解一定不会出现在 (i_2, j_2) ，则称 (i_1, j_1) 优于 (i_2, j_2) 。

Problem: 是否存在对称的条件使得搜索方向为主对角线？

2.2 复杂度分析

对于搜索空间 $[a, n] \times [b, m]$: 若 $Q(a, m) = 1$, 则搜索空间变为 $(a, m) \cup [a + 1, n] \times [b, m]$.

若 $Q(a, m) = 2$, 则搜索空间变为 $(a, m) \cup [a, n] \times [b, m - 1]$.

迭代直到解空间为 1.

复杂度 $O(n + m)$.

2.3 例题

2.3.1 力扣 11

$Q(i, j) = h_i > h_j$, 可证满足条件。

```

1  class Solution {
2  public:
3      int maxArea(vector<int>& height) {
4          int l = 0;
5          int r = height.size() - 1;
6          int maxas = 0;
7          while(l<=r){
8              maxas = max(maxas, (r - l) * min(height[l],height[r
9                  ]));
10             if(height[r] > height[l]){
11                 l++;
12             }
13             else{
14                 r --;
15             }
16             return maxas;
17         }
18     };

```

参考资料

[1]. [github-Competitive Programming](#)

[2]. [C++ 二分查找库函数 lower_bound, upper_bound, binary_search 的简单使用](#)

[3]. [力扣](#)

[4]. [单调函数](#)

数据结构-树上问题

ZeitHaum

2023 年 4 月 16 日

目录

1 基本概念	1
2 基础例题	1
2.1 例 1	1
2.2 例 2	2
2.3 例 3	3
2.4 例 4	4
2.5 例 5	6
3 *Morris 遍历	7
4 树的直径	7
4.1 算法一:DP	8
4.2 算法二: 遍历	8
4.3 适用性	9
4.4 例 6	9
4.4.1 Solution	9
4.4.2 Code	9
4.5 例 7	10
4.5.1 Solution	10
4.5.2 Code	10
4.6 例 8	12
4.6.1 Code	13
5 最近公共祖先问题	18
5.1 朴素算法	18
5.1.1 例 9	18
5.2 倍增算法	20
5.3 子问题一: 二进制转换	20
5.4 思路	22
5.5 例 11	23
6 线段树	24

6.1 概述	24
6.2 操作需要满足的条件	24
6.3 结构	25
6.4 性质	25
6.5 PURQ 数据结构	26
6.5.1 例 13	27
6.6 RURQ 数据结构	29
6.6.1 例 14	29
6.7 例 15	32
6.8 线段树求解 lca 问题	35
6.8.1 子问题:RMQ 问题	35
6.8.2 例 16	35
6.8.3 思路	37
6.8.4 例 17	38
7 树链剖分	41
7.1 概述	41
7.2 数据结构定义	41
7.3 性质	41
7.4 树链剖分解决 lca 问题	42
7.4.1 思路	42
7.4.2 复杂度分析	42
7.4.3 例 12	43
7.5 树链剖分与线段树	45
7.5.1 例 18	45
8 参考资料	49

1 基本概念

1. 无根树: 没有固定根节点的树。
2. 有根树: 指定固定根节点的无根树。
3. 深度: 结点到根节点的边数, 记作 $h(v)$ 。
4. 高度: 一个树中所有结点的深度的最大值, 记作 $h(T)$ 。
5. 完整二叉树: 每个结点的儿子数量要么为 0 要么为 1。
6. 完美二叉树: 所有叶节点深度均相同的完整二叉树。
7. 完全二叉树: 除了右下连续部分叶节点深度为树的高度减 1 外, 其余叶节点深度均和树的高度相同的二叉树。
8. 左孩子右兄弟: 两个数组分别记录每个结点的最左儿子和右兄弟。
9. 最近公共祖先: 记为 $\text{lca}(u, v)$ 。

2 基础例题

热身运动开始。

2.1 例 1

[力扣-94. 二叉树的中序遍历](#) 前序遍历、后序遍历类似, 不过需要调换添加答案的时机。

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *     {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *     ), left(left), right(right) {}
```

```

10  * };
11  */
12  class Solution {
13  public:
14      vector<int>ans;
15
16      void dfs(TreeNode* root){
17          if(root==nullptr) return;
18          dfs(root->left);
19          ans.push_back(root->val);
20          dfs(root->right);
21      }
22
23      vector<int> inorderTraversal(TreeNode* root) {
24          ans.clear();
25          dfs(root);
26          return ans;
27      }
28  };

```

2.2 例 2

力扣-100. 相同的树 同步 DFS 遍历步骤并遇到异常及时处理即可。

```

1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *         {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *         ), left(left), right(right) {}
12    * };
13    */
14    class Solution {
15    public:

```



```

14     bool check = true;
15     void dfs_same(TreeNode* r1,TreeNode* r2){
16         if(check==false) return;
17         if(r1==nullptr || r2==nullptr){
18             if(!(r1 == nullptr && r2==nullptr)) check = false;
19             return;
20         }
21         if(r1->val!=r2->val) check = false;
22         dfs_same(r1->left,r2->left);
23         dfs_same(r1->right,r2->right);
24     }
25
26     bool isSameTree(TreeNode* p, TreeNode* q) {
27         check = true;
28         dfs_same(p,q);
29         return check;
30     }
31 };

```

2.3 例 3

力扣-101. 对称二叉树 类似与例 2，不过同步遍历时考虑对称性即可。

```

1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *     {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *     ), left(left), right(right) {}
12    * };
13    */
14    class Solution {
15    public:
16        bool ck = true;

```

```

15 void dfs_symmetric(TreeNode* r1,TreeNode* r2){
16     if(r1==nullptr || r2==nullptr){
17         if(!(r1==nullptr && r2 == nullptr)) ck = false;
18         return;
19     }
20     if(r1->val!=r2->val) ck = false;
21     if(ck==false) return;
22     dfs_symmetric(r1->left,r2->right);
23     dfs_symmetric(r1->right,r2->left);
24 }
25
26 bool isSymmetric(TreeNode* root) {
27     ck = true;
28     if(root==nullptr) return ck;
29     dfs_symmetric(root->left,root->right);
30     return ck;
31 }
32 };

```

2.4 例 4

力扣-105. 从前序与中序遍历序列构造二叉树 分治和递归思想的应用。

```

1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8  *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9  *     {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *     ), left(left), right(right) {}
12    * };
13    */
14 class Solution {
15 public:
16     int n;

```

```

15     map<int,int>f;
16
17     TreeNode* buildTree(vector<int>& preorder, vector<int>&
18         inorder) {
19         function<void(int,int,int,int,TreeNode*)> dfs_build =
20             [&](int pbegin,int pend,int ibegin,int iend,
21                 TreeNode* root){
22                 //左闭右开
23                 int i_ind = f[root->val];
24                 int left_len = i_ind-1 - ibegin + 1;
25                 int right_len = iend-1 - (i_ind+1) + 1;
26                 if(left_len!=0){
27                     int root_val = preorder[pbegin+1];
28                     TreeNode* left = new TreeNode(root_val);
29                     root->left = left;
30                     dfs_build(pbegin+1,pbegin+1+left_len,ibegin,
31                         i_ind,left);
32                 }
33                 if(right_len!=0){
34                     int root_val = preorder[pbegin+1+left_len];
35                     TreeNode* right = new TreeNode(root_val);
36                     root->right = right;
37                     dfs_build(pbegin+1+left_len,pend,i_ind+1,iend,
38                         right);
39                 }
40             };
41             n = preorder.size();
42             f.clear();
43             map<int,int>g;
44             for(int i = 0;i<n;i++){
45                 g[inorder[i]] = i;
46             }
47             for(int i = 0;i<n;i++){
48                 f[preorder[i]] = g[preorder[i]];
49             }
50             TreeNode* root = new TreeNode(preorder[0]);
51             dfs_build(0,n,0,n,root);
52             return root;
53     }

```

```
49 };
```

2.5 例 5

力扣-106. 从中序与后序遍历序列构造二叉树 与上一题类似，注意递归条件的更改（前序遍历和后序遍历的区别）。

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *         {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *         ), left(left), right(right) {}
12    * };
13    */
14    class Solution {
15    public:
16        int n;
17        map<int,int>f;
18        TreeNode* buildTree(vector<int>& inorder, vector<int>&
19        postorder) {
20            //declaration of dfs.
21            function<void(int,int,int,int,TreeNode*)> dfs_build =
22            [&](int ibegin,int iend,int pbegin,int pend,
23            TreeNode* root){
24                int i_ind = f[root->val];
25                int left_len = i_ind-1 - ibegin + 1;
26                int right_len = iend-1 - (i_ind+1) + 1;
27                if(left_len!=0){
28                    TreeNode* left = new TreeNode(postorder[pbegin+
29                    left_len-1]);
30                    root->left = left;
31                    dfs_build(ibegin,i_ind,pbegin,pbegin+left_len,
32                    left);
```

```

26         }
27         if(right_len!=0){
28             TreeNode* right = new TreeNode(postorder[pend
29                 -2]);
30             root->right = right;
31             dfs_build(i_ind+1, iend, pbegin+left_len, pend-1,
32                 right);
33         }
34     };
35     //initialize of attributes.
36     n = inorder.size();
37     f.clear();
38     map<int,int>g;
39     for(int i = 0;i<n;i++){
40         g[inorder[i]] = i;
41     }
42     for(int i = 0;i<n;i++){
43         f[postorder[i]] = g[postorder[i]];
44     }
45     //calc ans.
46     TreeNode* root = new TreeNode(postorder[n-1]);
47     dfs_build(0,n,0,n,root);
48     return root;
49 }
50 };

```

3 *Morris 遍历

通过修改原始数据结构实现的空间 $O(1)$ 的算法。相比 DFS 和 BFS 空间 $O(n)$ 的算法空间效率很高，但是由于会修改原始数据结构不常用。

4 树的直径

定义为树 T 的最长路径，记为 $D(T)$ 。

4.1 算法一:DP

设 T_l, T_r 为左子树和右子树。考虑根节点对答案的贡献, 可以得到转移方程

$$D(T) = \max(D(T_l), D(T_r), h(T_l) + h(T_r)) \quad (1)$$

时间复杂度 $O(n)$, 如果还要得到一条直径的两个端点或者路径, 还需要记录取最大时的信息。

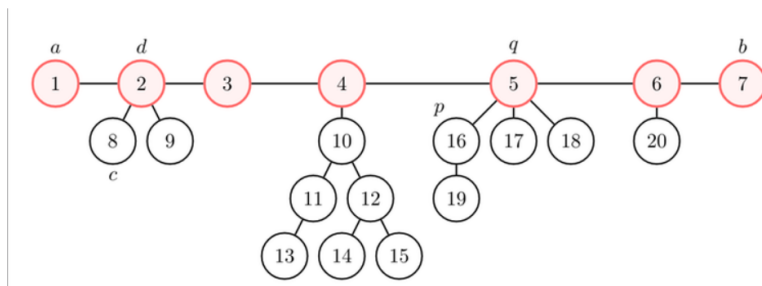
4.2 算法二: 遍历

记 $\text{dis}(u, v)$ 为结点 u 到结点 v 的距离。

引理 1 对于树上任意一个结点 x , 记 $a = \max_i(\text{dis}(x, i))$, 则 a 必为直径端点之一。

证明:

首先一棵树可以视为直径 + 森林, 如下图 (图片仅作参考):



而森林的每个顶点均在直径上。设 x 所在部分的顶点为 r_1 。

(利用反证法) 假设 a 不为直径端点, s, t 为一条直径的两个端点。

不失一般性地可设 $\text{dis}(x, s) \leq \text{dis}(x, t)$ 。

则有

$$\text{dis}(x, a) > \text{dis}(x, t) \Rightarrow \text{dis}(r_1, a) > \text{dis}(r_1, t) \Rightarrow \text{dis}(s, a) > \text{dis}(s, t). \quad (2)$$

这与假设相悖, 于是原命题得证。

同样根据定义可证以下引理:

引理 2 对于树上一个直径的端点 a , 记 $b = \max_i(\text{dis}(a, i))$, 则 $a \rightarrow b$ 的路径必为直径。

于是只需按以上两个引理操作，分别以任意结点和第一次的结果进行两次树的遍历（推荐 BFS 遍历），每次寻找到距离根节点最远的结点，第二次的结果即是答案。

4.3 适用性

当题目所给为典型有根树状结构时，适合用算法一。如果是无根树（图论结构），适合用算法二。

4.4 例 6

[力扣-543. 二叉树的直径](#)

4.4.1 Solution

典型例题，给的数据结构为树，适用于算法一。

4.4.2 Code

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *         {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *         ), left(left), right(right) {}
12    * };
13    */
14    //DP version
15    class Solution {
16    public:
17        int ans = 0;
18        int dfs_dp(TreeNode* root){
19            if(root==nullptr) return 0;
```

```

18     int h_l = dfs_dp(root->left);
19     int h_r = dfs_dp(root->right);
20     ans = max(ans, h_l+h_r);
21     return max(h_l, h_r)+1;
22 }
23
24 int diameterOfBinaryTree(TreeNode* root) {
25     dfs_dp(root);
26     return ans;
27 }
28 };

```

4.5 例 7

[CodeForces-Round 862\(Div2\)-D](#)

4.5.1 Solution

此题较为灵活，分析可得显然直径的端点是首次被合入的对象。设 T_i 表示以 i 为根时的树。

显然第 k 轮将会合并满足 $h(T_i) = k$ 的点，即连通区域减少 $\text{count}(h(T_i) = k)$ (第一次合并减少 $\text{count}(h(T_i) = k) - 1$)。

根据引理1, $h(T_i) = k$ 等价于距离直径一端点距离较大值为 k 。

也即 $\max(\text{dis}(i, s), \text{dis}(i, t)) = k$ 。

于是我们可以先找到一组直径的端点 s, t ，即可计算 $h(T_i)$ 。

根据所给条件适宜使用算法二。

4.5.2 Code

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 ostream& operator<<(ostream& out, vector<int>& arr){
5     for(int i = 0; i<arr.size(); i++){
6         out<<arr[i]<<" ";
7     }
8     return out;

```



```

9  }
10
11 int main(){
12     int n;
13     cin>>n;
14     vector adj(n+1,vector<int>(0));
15     int u,v;
16     for(int i = 0;i<n-1;i++){
17         cin>>u>>v;
18         adj[u].push_back(v);
19         adj[v].push_back(u);
20     }
21     vector dis_1(n+1,-1);
22     vector dis_2(n+1,-1);
23     auto bfs = [&](vector<int>& height,int root){
24         queue<int> q;
25         q.push(root);
26         height[root] = 0;
27         vector<bool>vis(n+1,false);
28         while(q.size()!=0){
29             int qf = q.front();
30             q.pop();
31             vis[qf] = true;
32             for(int i = 0;i<adj[qf].size();i++){
33                 int now = adj[qf][i];
34                 if(!vis[now]){
35                     q.push(now);
36                     height[now] = height[qf]+1;
37                     vis[now] = true;
38                 }
39             }
40         }
41     };
42     bfs(dis_2,1);
43     int max_root = max_element(dis_2.begin(),dis_2.end()) -
        dis_2.begin();
44     bfs(dis_1,max_root);
45     max_root = max_element(dis_1.begin(),dis_1.end()) - dis_1.
        begin();

```

```

46     bfs(dis_2,max_root);
47     for(int i = 0;i<dis_1.size();i++){
48         dis_1[i] = max(dis_1[i],dis_2[i]);
49     }
50     sort(dis_1.begin(),dis_1.end());
51     vector ans(n+1,0);
52     for(int i =1;i<dis_1.size();i++){
53         ans[dis_1[i]]++;
54     }
55     cerr<<ans<<endl;
56     ans[n] = n;
57     bool enable = false;
58     for(int i = ans.size()-2;i>=0;i--){
59         int temp = 0;
60         if(ans[i]!=0 && !enable){
61             temp = 1;
62             enable = true;
63         }
64         ans[i] = ans[i+1] - ans[i] + temp;
65     }
66     for(int i = 1;i<ans.size();i++){
67         cout<<ans[i]<<" ";
68     }
69     cout<<endl;
70 }

```

4.6 例 8

[洛谷-SDOI2013-直径](#) 这道题是上述引理的综合应用。

第一问略。第二问需要求取所有直径都经过的边数，为此我们可以先将一条直径求取出来。

(利用树 = 直径 + 森林的做法) 记 s, t 为一条直径的两个端点， r 是直径上一点， T_r 是去除直径后以 r 为根节点的深度。

显然，当 $\exists x$ 使得在树 T_r 中有 $h(x) = \text{dis}(x, s)$ 时，路径 $s \rightarrow r$ 便不是所有直径都经过的边的一部分，对于端点 t 此结论也成立。

引理 3 对于上述描述的 x , 若存在, 必有

$$h(x) = h(T_r) = \min(\text{dis}(r, s), \text{dis}(r, t)). \quad (3)$$

证明: 使用反证法, 证明如果不满足上述情况必有假设 “ s, t 为直径的两个端点” 断言为假即可。

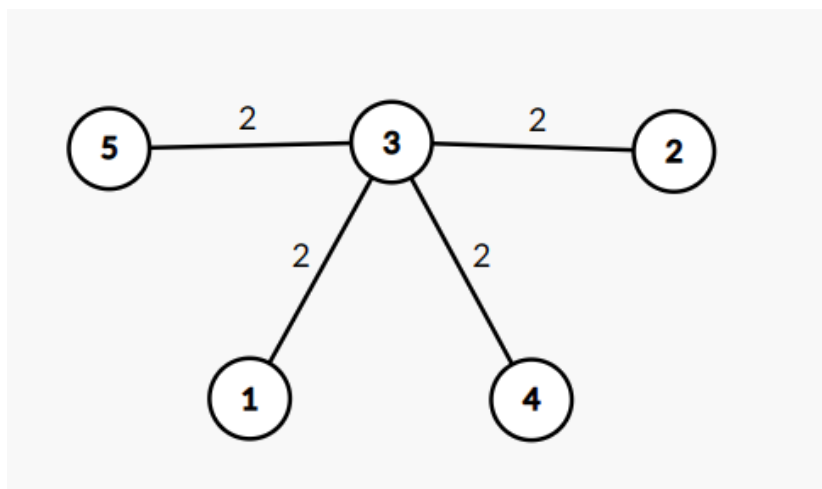
记

$$f(v) = \max_{\text{dis}(u, v) = h(T_u)} (\text{dis}(u, v)), v \in \{s, t\}. \quad (4)$$

于是

$$\text{ans} = \text{abs}(f(s) - f(t)). \quad (5)$$

上述算法在下图将会遇到特例, 需要进行特别判定:



4.6.1 Code

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 #define int long long
4 #define mp(x,y) make_pair((x),(y))
5
6 ostream& operator<<(ostream& out, const vector<int>& arr){
7     for(int i = 0; i < arr.size(); i++){
8         out << arr[i] << " ";
9     }

```

```

10     return out;
11 }
12 ostream& operator<<(ostream& out,const vector<bool>& arr){
13     for(int i = 0;i<arr.size();i++){
14         out<<arr[i]<<" ";
15     }
16     return out;
17 }
18 ostream& operator<<(ostream& out,const pair<int,int>& p){
19     out<<p.first<<","<<p.second<<" ";
20     return out;
21 }
22 ostream& operator<<(ostream& out,const map<pair<int,int>,int>&
    dic){
23     for(auto iter = dic.begin();iter!=dic.end();iter++){
24         out<<iter->first<<":"<<iter->second<<" ";
25     }
26     return out;
27 }
28
29 signed main(){
30     int n;
31     cin>>n;
32     //handling the input.
33     const int ARR_SIZE = n+1;
34     vector adj(ARR_SIZE,vector<int>(0));
35     map<pair<int,int>,int>weight;
36     int u,v,w;
37     for(int i = 0;i<n-1;i++){
38         cin>>u>>v>>w;
39         adj[u].push_back(v);
40         adj[v].push_back(u);
41         weight[mp(u,v)] = w;
42         weight[mp(v,u)] = w;
43     }
44     //Find a diameter point.
45     vector<int>h(ARR_SIZE,0);
46     vector<bool>vis(ARR_SIZE,false);
47     auto clear_vis = [&]{

```

```

48         for(int i = 0;i<vis.size();i++){
49             vis[i] = false;
50             h[i] = 0;
51         }
52     };
53     auto clear_h = [&]{
54         for(int i = 0;i<vis.size();i++){
55             h[i] = 0;
56         }
57     };
58     auto clear_vis_h = [&]{
59         clear_vis();
60         clear_h();
61     };
62     auto bfs_find = [&](vector<int>& h,int root){
63         queue<int> q;
64         q.push(root);
65         vis[root] = true;
66         while(q.size()>0){
67             int qf = q.front();
68             q.pop();
69             for(int i = 0;i<adj[qf].size();i++){
70                 int now = adj[qf][i];
71                 if(vis[now]==false){
72                     q.push(now);
73                     vis[now] = true;
74                     h[now] = h[qf] + weight[mp(qf,now)];
75                 }
76             }
77         }
78         return max_element(h.begin(),h.end());
79     };
80     clear_vis_h();
81     int da = bfs_find(h,1) - h.begin();
82     clear_vis_h();
83     int db = bfs_find(h,da) - h.begin();
84     int d_len = h[db];
85     //Find a diameter path.
86     vector<int> diameter_path(0);

```

```

87     for(int i = 0;i<vis.size();i++){
88         vis[i] = false;
89     }
90     bool enable_path = false;
91     function<void(int,int)> dfs_path = [&](int root,int
        now_weight){
92         diameter_path.push_back(root);
93         vis[root] = true;
94         for(int i = 0;i<adj[root].size();i++){
95             int now = adj[root][i];
96             if(vis[now]==false){
97                 if(now_weight+ weight[mp(root,now)]==d_len &&
                    adj[now].size()==1){
98                     diameter_path.push_back(now);
99                     enable_path = true;
100                     return;
101                 }
102                 dfs_path(now,now_weight+weight[mp(root,now)]);
103                 if(enable_path) return;
104                 diameter_path.pop_back();
105             }
106         }
107     };
108     dfs_path(da,0);
109     int d_size = diameter_path.size();
110     //Check connectivity.
111     for(int i = 0;i<vis.size();i++){
112         vis[i] = false;
113     }
114     for(int i =0;i<d_size;i++){
115         vis[diameter_path[i]] = true;
116     }
117     int temp_h;
118     vector<int> t_r(d_size,0);
119     function<void(int,int)> dfs_check = [&](int root,int j){
120         vis[root] = true;
121         for(int i = 0;i<adj[root].size();i++){
122             int now = adj[root][i];
123             if(vis[now]==false){

```

```

124         temp_h += weight[mp(root,now)];
125         //Leaf Node;
126         if(adj[now].size()==1){
127             t_r[j] = max(t_r[j],temp_h);
128         }
129         dfs_check(now,j);
130         temp_h -= weight[mp(root,now)];
131     }
132 }
133 };
134 for(int i = 0;i<d_size;i++){
135     temp_h = 0;
136     dfs_check(diameter_path[i],i);
137 }
138 int fs = max_element(t_r.begin(),t_r.begin()+(d_size-1)
139                     /2+1) - t_r.begin();
140 int ft = max_element(t_r.begin()+(d_size-1)/2+1,t_r.end())
141         - t_r.begin();
142 int ans2 = abs(fs - ft);
143 //Special Judge
144 int accu_w = 0;
145 for(int i = 1;i<d_size;i++){
146     accu_w += weight[mp(diameter_path[i-1],diameter_path[i]
147                        )]];
148     if(d_len - accu_w==accu_w){
149         if(t_r[i]==accu_w) ans2 = 0;
150         break;
151     }
152 }
153 cout<<d_len<<endl;
154 cout<<ans2<<endl;
155 }
156 /*
157 5
158 1 3 2
159 2 3 2
160 4 3 2
161 5 3 2

```

```

160 | 9
161 | 1 2 1
162 | 2 5 3
163 | 3 4 2
164 | 4 5 1
165 | 5 6 1
166 | 6 7 3
167 | 5 8 2
168 | 8 9 2
169 | */

```

5 最近公共祖先问题

两个结点的最近公共祖先为公共结点中深度最低的结点, 记为 $\text{lca}(u, v)$ 。

性质:

1. $\text{lca}(u, v) = u$, 当且仅当 u 是 v 的祖先。
2. 对于点集 A, B , $\text{lca}(A \cup B) = \text{lca}(\text{lca}(A), \text{lca}(B))$.
3. $\text{dis}(u, v) = h(u) + h(v) - 2 * h(\text{lca}(u, v))$.

5.1 朴素算法

记 $\text{fa}(x)$ 为结点 x 的根节点, 根据性质 1, 可以推出转移方程式:

$$\text{lca}(u, v) = \begin{cases} u, & u = v \\ \text{lca}(\text{fa}(u), v), & h(u) > h(v) \\ \text{lca}(u, \text{fa}(v)), & h(u) < h(v) \\ \text{lca}(\text{fa}(u), \text{fa}(v)), & h(u) = h(v) \end{cases} \quad (6)$$

预处理复杂度 $O(n)$, 单次查询复杂度为 $O(h(T))$ 。

5.1.1 例 9

[力扣-236. 二叉树的最近公共祖先](#)

Code


```

1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8   * };
9   */
10 class Solution {
11 public:
12     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
13         TreeNode* q) {
14         map<int,TreeNode*> fa;
15         map<int,int> h;
16         auto bfs = [&]() {
17             queue<TreeNode*> q;
18             q.push(root);
19             fa[root->val] = NULL;
20             h[root->val] = 0;
21             while(q.size() != 0) {
22                 TreeNode* qf = q.front();
23                 q.pop();
24                 if(qf->left != NULL) {
25                     fa[qf->left->val] = qf;
26                     h[qf->left->val] = h[qf->val] + 1;
27                     q.push(qf->left);
28                 }
29                 if(qf->right != NULL) {
30                     fa[qf->right->val] = qf;
31                     h[qf->right->val] = h[qf->val] + 1;
32                     q.push(qf->right);
33                 }
34             }
35         };
36         bfs();
37         function<TreeNode*(TreeNode*,TreeNode*)> find_lca =
38             [&](TreeNode* p,TreeNode* q) {

```

```

37         if(p->val==q->val) return p;
38         if(h[p->val]>h[q->val]) return find_lca(fa[p->val],
           q);
39         else if(h[p->val]<h[q->val]) return find_lca(p,fa[q
           ->val]);
40         else return find_lca(fa[p->val],fa[q->val]);
41     };
42     return find_lca(p,q);
43 }
44 };

```

5.2 倍增算法

5.3 子问题一: 二进制转换

首先先思考一个子问题: 对于一个整数 x , 如何将其转换为二进制?

对于这个问题, 朴素的做法是依次除 2, 即利用表达式

$$\text{binary_str}(x) = \begin{cases} \text{binary_str}(x/2) + '0', & x \% 2 = 0 \\ \text{binary_str}(x/2) + '1', & x \% 2 = 1 \end{cases} \quad (7)$$

实践中, 为了避免从字符串首部插入导致复杂度提升, 常常先将字符插入至尾部, 最后再进行翻转。尽管如此, 单次求取 $x/2$ 常数还是比较大 (较比于加减法和乘法)。

如果我们预处理数列 $\{2^0, 2^1, \dots, 2^k\}, k \leq \lfloor \log_2(n) \rfloor$ 的值, 于是可有

$$\text{binary_str}(x) = \begin{cases} '0' + \text{binary_str}(x), & x < 2^k \\ '1' + \text{binary_str}(x - 2^k), & x \geq 2^k \end{cases} \quad (8)$$

依次减少 k 的值, 便可以求出二进制每位的值。

例 10: 转换二进制

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int pow2[31];
5
6 string binary_str1(int x){

```

```
7     string rans; //初始设置ans为答案的翻转串
8     while(true){
9         if(x%2==0) rans.push_back('0');
10        else rans.push_back('1');
11        x = x/2;
12        if(x==0) break;
13    }
14    reverse(rans.begin(),rans.end());
15    return rans;
16 }
17
18 string binary_str2(int x){
19     string ans;
20     for(int i = 30;i>=0;i--){
21         if(x>= pow2[i]){
22             x -= pow2[i];
23             ans.push_back('1');
24         }
25         else ans.push_back('0');
26     }
27     //如果要求去除前导0...
28     int first_1 = 0;
29     for(int i = 1;i<=31;i++){
30         if(ans[i]=='1'){
31             first_1 = i;
32             break;
33         }
34     }
35     return ans.substr(first_1,31 - first_1+1);
36 }
37
38 int main(){
39     //预处理pow2
40     for(int i = 0;i<31;i++){
41         if(i==0) pow2[i] = 1;
42         else pow2[i] = pow2[i-1] * 2;
43     }
44     int x = 13;
45     cout<<binary_str1(x)<<endl<<binary_str2(x)<<endl;
```

5.4 思路

上述子问题给出了一种可能性, 对于具备二分性质的数列可以通过预处理 2^k 数列来降低多次询问复杂度。

现在我们考虑利用这种方法优化朴素最近公共祖先问题。

首先我们考虑当 $h(u) \neq h(v)$ 的情况, 不失一般性地假设 $h(u) > h(v)$, 于是必存在 x 使得 $h(\text{fa}^x(u)) = h(v)$, 其中

$$\text{fa}^x(u) = \begin{cases} \text{fa}^{x-1}(u), & x > 1 \\ \text{fa}(u), & x = 1 \\ u, & x = 0. \end{cases} \quad (9)$$

对于任意 k , 根据二进制分解的原理有

$$\text{fa}^x(u) = \begin{cases} \text{fa}^x(u), & h(\text{fa}^{2^k}(u)) < h(u) \\ \text{fa}^{x-2^k}(\text{fa}^{2^k}(u)), & h(\text{fa}^{2^k}(u)) \geq h(u) \end{cases} \quad (10)$$

如果对于任意结点 u 我们提前预处理 $\text{fa}^{2^k}(u)$ 的值, 然后依次减小 k 的值即可。

单次查询复杂度: $O(\log(n))$, k 最大为 $\lfloor \log(n) \rfloor$ 。

现在我们考虑当 $h(u) = h(v)$ 时求最近公共祖先的操作, 同理, 必存在最小的 y , 使得 $\text{fa}^y(u) = \text{fa}^y(v)$ 。

同理, 根据二进制分解的原理, 对于 k 有

$$\text{lca}(u, v) = \begin{cases} \text{lca}(u, v), & \text{fa}^{2^k}(u) = \text{fa}^{2^k}(v) \\ \text{lca}(\text{fa}^{2^k}(u), \text{fa}^{2^k}(v)), & \text{fa}^{2^k}(u) \neq \text{fa}^{2^k}(v) \end{cases} \quad (11)$$

最后需要考虑 $\text{fa}^{2^k}(u)$ 的维护:

$$\text{fa}^{2^k}(u) = \begin{cases} \text{fa}(u), & k = 0 \\ \text{fa}^{2^{k-1}}(\text{fa}^{2^{k-1}}(u)), & k \geq 1. \end{cases} \quad (12)$$

注意需要考虑对非法值, 即 $2^k > h(u)$ 时的处理, 一般考虑赋为定义域之外的数。

5.5 例 11

P3379 【模板】最近公共祖先 (LCA) 模板题，按照模板写即可。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int MAX_SIZE = 5e5+1;
5  int fa[MAX_SIZE][21];
6  int h[MAX_SIZE];
7
8  int main(){
9      ios::sync_with_stdio(false);
10     cin.tie(0);
11     int n,m,root;
12     cin>>n>>m>>root;
13     vector adj(n+1,vector<int>(0));
14     int u,v;
15     for(int i = 0;i<n-1;i++){
16         cin>>u>>v;
17         adj[u].push_back(v);
18         adj[v].push_back(u);
19     }
20     auto dfs = [&](auto self,int root,int depth)->void{
21         for(int v: adj[root]){
22             if(v==fa[root][0]) continue;
23             fa[v][0] = root;
24             h[v] = depth+1;
25             self(self,v,depth+1);
26         }
27     };
28     dfs(dfs,root,0);
29     h[0] = -1;
30     for(int i = 1;i<21;i++){
31         for(int j = 1;j<=n;j++){
32             fa[j][i] = fa[fa[j][i-1]][i-1];
33         }
34     }
35     auto lca = [&](int u,int v){
36         if(h[u]<h[v]) swap(u,v);

```

```

37         if(h[u]!=h[v]){
38             for(int i = 20;i>=0;i--){
39                 if(h[fa[u][i]]>=h[v]) u = fa[u][i];
40             }
41         }
42         if(u==v) return u;
43         else{
44             for(int i = 20;i>=0;i--){
45                 if(fa[u][i] != fa[v][i]){
46                     u = fa[u][i];
47                     v = fa[v][i];
48                 }
49             }
50             return fa[u][0];
51         }
52     };
53     for(int i = 0;i<m;i++){
54         cin>>u>>v;
55         cout<<lca(u,v)<<endl;
56     }
57 }

```

6 线段树

6.1 概述

定义: 将区间信息存储在结点的一种数据结构。其支持的操作为

1. 区间修改, 将区间 $[L, R]$ 修改为新值。
2. 区间查询, 查询区间 $[L, R]$ 的信息。

6.2 操作需要满足的条件

线段树基于分治的原理。其维护的信息 (统称操作) 都需要满足以下条件:

1. 操作必须是二元的。

2. 操作必须满足结合律。
3. 操作必须有么元。

几个满足线段树操作的例子: 加减法、最大值、最小值、GCD、LCM 等。

6.3 结构

常用的线段树满足以下性质:

1. 线段树是一颗完整二叉树。
2. 线段树的根节点编号为 1, 维护区间 $[1, n]$ 的信息。
3. 对于区间 $[1, n]$, 线段树的叶子结点从左往右依次维护 $[1, 1], [2, 2], \dots, [n-1, n-1], [n, n]$ 的信息。
4. 对于线段树的每个中间结点, 设该节点维护区间 $[x, y]$ 的信息, 则其左儿子维护区间 $[x, \lfloor \frac{x+y}{2} \rfloor]$ 的信息, 右儿子维护区间 $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$ 的信息。
5. 线段树采取堆式存储, 也即对于编号 x , 其左儿子编号为 $2x$, 右儿子编号为 $2x + 1$ 。

6.4 性质

注意: 这里的 n 指的是区间长度而不是常见的树的结点个数。

引理 4 线段树不存在两个结点不同但编号一样。

证明: 假设存在两个结点编号均为 x , 所以其均为 $\lfloor x/2 \rfloor$ 的子节点, 且根据奇偶性可以判定其属于左儿子还是右儿子。

因此这两个结点是同一个结点, QED。

引理 5 线段树的结点个数小于等于 $2n - 1$ 。

证明:

观察可得递推式

$$f(n) = \begin{cases} 2f(n/2) + 1, & n \text{ 是奇数,} \\ f((n-1)/2) + f((n+1)/2) + 1, & n \text{ 是偶数.} \end{cases}$$

使用归纳法证明。当 $f(1) = 1, f(2) = 3$ 均成立。

考虑 $f(n)$,

当 n 为偶数时, $f(n) \leq 2 \times (n/2 \times 2 - 1) + 1 \Leftrightarrow f(n) \leq 2n - 1$.

当 n 为奇数时, $f(n) \leq (n-1)/2 \times 2 - 1 + (n+1)/2 \times 2 - 1 + 1 \Leftrightarrow f(n) \leq 2n - 1$.

均成立, QED.

引理 6 线段树编号最大值小于 $4n$ 。

证明:

线段树深度最大不超过 $2^{\lceil \log(n) \rceil}$, 因此最大编号不超过

$$1 + 2 + 4 + \dots + 2^{\lceil \log(n) \rceil}$$

即为 $2^{\lceil \log(n) \rceil + 1} < 2^{\log(n) + 2} = 4n$.

引理 7 线段树的高度不超过 $O(\log(n))$.

证明: 由性质5和完全二叉树的性质可证。-

6.5 PURQ 数据结构

PURQ —— "Point update range query", 也即单点修改, 区间查询。

首先要构建线段树, 开 4 倍空间采取递归建树即可。

其次是区间查询,

设 e 为查询信息的么元, \circ 为查询信息合并的操作。

要查询的区间为 $[L, R]$, 当前所在叶节点维护的区间为 $[l, r]$, 信息为 $\text{inf}(l, r)$.

有以下递推式:

$$Q(L, R, l, r) = \begin{cases} \text{inf}(l, r), & [l, r] \subseteq [L, R] \\ e, & [l, r] \cap [L, R] = \emptyset \\ Q(L, R, l, (l+r)/2) \circ Q(L, R, (l+r)/2 + 1, r), & \text{else} \end{cases}$$

以上三种情况的条件也称为完全覆盖、不相交、部分覆盖。

单点更新和区间查询类似, 见源代码。

6.5.1 例 13

模板题，需要注意区间关系的判断条件。[力扣-307. 区域和检索 - 数组可修改](#)

```

1  class NumArray {
2  public:
3      vector<int>s;
4      int n;
5
6      NumArray(vector<int>& nums) {
7          n = nums.size();
8          s.resize(n*4+1);
9          nums.insert(nums.begin(),0);
10         auto build = [&](auto self,int l,int r,int p)->void{
11             if(l==r){
12                 s[p] = nums[l];
13                 return;
14             }
15             int mid = (l+r)/2;
16             self(self,l,mid,p*2);
17             self(self,mid+1,r,p*2+1);
18             s[p] = s[p*2] + s[p*2+1];
19         };
20         build(build,1,n,1);
21     }
22
23     void update(int index, int val) {
24         index = index + 1;
25         auto up = [&](auto self,int l,int r,int p)->void{
26             if(l<=index && r>=index){
27                 if(l==r){
28                     s[p] = val;
29                     return;
30                 }
31                 int mid = (l+r)/2;
32                 self(self,l,mid,p*2);
33                 self(self,mid+1,r,p*2+1);
34                 s[p] = s[p*2] + s[p*2+1];
35                 return;

```

```

36         }
37         else{
38             return;
39         }
40     };
41     up(up,1,n,1);
42 }
43
44 int sumRange(int left, int right) {
45     left = left + 1;
46     right = right + 1;
47     auto query = [&](auto self, int l,int r,int p)->int{
48         if(left<=l && right>= r){
49             return s[p];
50         }
51         else if(l>right || r<left){
52             return 0;
53         }
54         else{
55             int mid = (l+r)/2;
56             return self(self,l,mid,p*2)+self(self,mid+1,r,p
                    *2+1);
57         }
58     };
59     return query(query,1,n,1);
60 }
61 };
62
63 /**
64  * Your NumArray object will be instantiated and called as such
65  * :
66  * NumArray* obj = new NumArray(nums);
67  * obj->update(index,val);
68  * int param_2 = obj->sumRange(left,right);
69  */

```

6.6 RURQ 数据结构

RURQ —— “Range update range query” 数据结构，即区间修改，区间查询。

思考区间修改的问题，如果每次都去修改区间内的数据将会复杂度难以承受。

而如果只保存此次操作，改变对应结点的信息，便可以大大降低复杂度。

懒惰标记 (lazy propagation) 是优化区间更新的方法，其思想便是存储每次操作的结果，数组的更新只在进入到相应区间时发生。

懒惰标记维护的操作集合 F 需要满足以下条件，

1. F 含有单位变换 e ，使得 $e(x) = x$ 。
2. F 中的元素是可复合的，也即对于 $f, g \in F$ ，必有 $f \circ g \in F$ 。
3. 对于维护信息的操作 \cdot ， F 中的元素是可分配的。也即对于 $f \in F$ ， $f(x \cdot y) = f(x) \cdot f(y)$ 。

具体编程实现见例题。

6.6.1 例 14

洛谷 - P3372 【模板】线段树 1

需要维护的操作为区间求和，记 $\text{lazy}(i)$ 表示线段树结点 i 未对子树更新部分。

一旦查询或更新到相应区间， $\text{lazy}(i)$ 存储的更新应该同步向下传递，而本结点的标记清空，并且递归结束后需要更新该区间维护的信息。

代码:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define MAX_N 100001
4 #define int long long
5
6 struct S{
7     int val;
8     int len;
9     S():val(0),len(0){}
```

```

10     S(int val,int len):val(val),len(len){}
11     S operator+(const S& s){
12         return {this->val+s.val,this->len+s.len};
13     }
14 };
15 int da[MAX_N];
16 S s[4*MAX_N];
17 const S se = {0,0};
18 int lazy[4*MAX_N];
19 struct seg_tree{
20     //将lazy 标记向下传递
21     void push_down(int p){
22         lazy[p*2] += lazy[p];
23         lazy[p*2+1] += lazy[p];
24         s[p*2].val += lazy[p]*s[p*2].len;
25         s[p*2+1].val += lazy[p]*s[p*2+1].len;
26         lazy[p] = 0;
27     }
28     //build
29     void build(int l,int r,int p){
30         if(l==r){
31             s[p] = {da[l],1};
32             return;
33         }
34         int mid = (l+r)/2;
35         build(l,mid,p*2);
36         build(mid+1,r,p*2+1);
37         s[p] = s[p*2] + s[p*2+1];
38     }
39     //query
40     S query(int l,int r,int p,int L,int R){
41         if(L<=l && R>=r) return s[p];
42         else if(l>R || r<L) return se;
43         else{
44             int mid = (l+r)/2;
45             push_down(p);
46             return query(l,mid,p*2,L,R) + query(mid+1,r,p*2+1,L
47                 ,R);

```

```

48     }
49     //update
50     void update(int l ,int r,int p,int L,int R,int val){
51         if(L<=l && R>=r) {
52             lazy[p] += val;
53             s[p].val += s[p].len*val;
54         }
55         else if(l>R || r<L) return;
56         else{
57             int mid = (l+r)/2;
58             push_down(p);
59             update(l,mid,p*2,L,R,val);
60             update(mid+1,r,p*2+1,L,R,val);
61             s[p] = s[p*2] + s[p*2+1];
62         }
63     }
64 };
65
66 signed main(){
67     ios::sync_with_stdio(false);
68     cin.tie(0);
69     int n,m;
70     cin>>n>>m;
71     seg_tree st;
72     for(int i = 1;i<=n;i++){
73         cin>>da[i];
74     }
75     st.build(1,n,1);
76     int type;
77     int L,R,val;
78     for(int i = 1;i<=m;i++){
79         cin>>type;
80         if(type==1){
81             cin>>L>>R>>val;
82             st.update(1,n,1,L,R,val);
83         }
84         else{
85             cin>>L>>R;
86             cout<<st.query(1,n,1,L,R).val<<endl;

```

```

87     }
88   }
89 }

```

6.7 例 15

洛谷 - P3373 【模板】线段树 2

与上述题目不同的是需要考虑操作的定义。

考虑操作 $f(a, b, x) = (x + a)b$ (先加后乘), 则

$$\begin{aligned}
 f(c, d, f(a, b, x)) &= (f(a, b, x) + c)d \\
 &= ((x + a)b + c)d \\
 &= (x + a + c/b)bd \\
 &= f(a + c/b, bd, x)
 \end{aligned}$$

若 $b = 0$, 此时便不满足区间更新所需的第二条性质。

尽管我们可以通过约定此时 $c/b = 0$ 使其满足该性质, 然而对于 c/b 这部分的小数仍然是我们不希望得到的 (不易维护)。

因此我们考虑定义 $f(a, b, x) = ax + b$ (先乘后加), 则

$$\begin{aligned}
 f(c, d, f(a, b, x)) &= f(c, d, ax + b) \\
 &= acx + bc + d \\
 &= f(ac, bc + d)
 \end{aligned}$$

此操作具备良好的性质, 便于维护。

代码:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define int long long
4  #define MAX_SIZE 100001
5
6  int p;
7  struct S
8  {
9      int val;
10     int len;

```

```

11     S():val(0),len(0){}
12     S(int val,int len):val(val),len(len){}
13     S merge(S s){ return {(this->val+s.val)%p,this->len+s.len
14         };}
15
16 struct F
17 {
18     int add;
19     int mul;
20     F():add(0),mul(1){}
21     F(int add,int mul):add(add),mul(mul){}
22     F composition(F f){ return {((this->add*f.mul)%p + f.add)%p
23         ,(this->mul*f.mul)%p};}
24 };
25 S apply(F f, S s){
26     return {(s.val * f.mul)%p + (s.len * f.add)%p,s.len};
27 }
28
29 int da[MAX_SIZE];
30 S s[MAX_SIZE*4];
31 F lazy[MAX_SIZE*4];
32 S se;
33 F fe;
34
35 struct seg_tree
36 {
37     void build(int l,int r,int p){
38         if(l==r){
39             s[p] = {da[l],1};
40             return;
41         }
42         int mid = (l+r)/2;
43         build(l,mid,p*2);
44         build(mid+1,r,p*2+1);
45         s[p] = s[p*2].merge(s[p*2+1]);
46     }
47

```

```

48     void push_down(int p){
49         lazy[p*2] = lazy[p*2].composition(lazy[p]);
50         lazy[p*2+1] = lazy[p*2+1].composition(lazy[p]);
51         s[p*2] = apply(lazy[p],s[p*2]);
52         s[p*2+1] = apply(lazy[p],s[p*2+1]);
53         lazy[p] = fe;
54     }
55
56     S query(int l,int r,int p,int L,int R){
57         if(L<= l && R>=r) return s[p];
58         else if(l>R || r<L) return se;
59         else{
60             int mid = (l+r)/2;
61             push_down(p);
62             return query(l,mid,p*2,L,R).merge(query(mid+1,r,p
               *2+1,L,R));
63         }
64     }
65
66     void update(int l,int r,int p,int L,int R,int add,int mul){
67         if(L<=l && R>=r){
68             lazy[p] = lazy[p].composition({add,mul});
69             s[p] = apply({add,mul},s[p]);
70         }
71         else if(l>R || r<L) return;
72         else{
73             int mid = (l+r)/2;
74             push_down(p);
75             update(l,mid,p*2,L,R,add,mul);
76             update(mid+1,r,p*2+1,L,R,add,mul);
77             s[p] = s[p*2].merge(s[p*2+1]);
78         }
79     }
80 };
81
82
83 signed main(){
84     ios::sync_with_stdio(false);
85     cin.tie(0);

```



```

86     int n,m;
87     cin>>n>>m>>p;
88     for(int i = 1;i<=n;i++){
89         cin>>da[i];
90     }
91     seg_tree st;
92     st.build(1,n,1);
93     int type,L,R,add,mul;
94     for(int i = 1;i<=m;i++){
95         cin>>type;
96         if(type==1){
97             cin>>L>>R>>mul;
98             st.update(1,n,1,L,R,0,mul);
99         }
100        else if(type==2){
101            cin>>L>>R>>add;
102            st.update(1,n,1,L,R,add,1);
103        }
104        else{
105            cin>>L>>R;
106            cout<<st.query(1,n,1,L,R).val<<endl;
107        }
108    }
109 }

```

6.8 线段树求解 lca 问题

6.8.1 子问题:RMQ 问题

RMQ——“Range Minimum/Maximum Query”，区间最大/最小值查询。

无修改，直接用 RQ 数据结构维护即可（甚至不需要 PU）。

6.8.2 例 16

板子题。Balanced Lineup

代码:

```

1 #include <iostream>

```

```

2  #include <algorithm>
3  #define MAX_SIZE 50001
4  #define N_INF -1000001
5  #define P_INF 1000001
6  using namespace std;
7
8  struct S
9  {
10     int mi;
11     int mx;
12     S():mi(P_INF),mx(N_INF){}
13     S(int mi,int mx):mi(mi),mx(mx){}
14 };
15
16 inline S merge(S s1, S s2){
17     S ret;
18     ret.mi = min(s1.mi,s2.mi);
19     ret.mx = max(s1.mx,s2.mx);
20     return ret;
21 }
22
23 int da[MAX_SIZE];
24 S s[4*MAX_SIZE];
25 S se;
26
27 struct seg_tree
28 {
29     void build(int l,int r,int p){
30         if(l==r){
31             s[p].mi = da[l];
32             s[p].mx = da[l];
33             return;
34         }
35         int mid = l+r>>1;
36         build(l,mid,p<<1);
37         build(mid+1,r,p<<1 | 1);
38         s[p] = merge(s[p<<1],s[p<<1 | 1]);
39     }
40

```

```

41     inline S query(int l,int r,int L,int R,int p){
42         if(L<= l && R>=r) return s[p];
43         else if(l>R || r<L) return se;
44         else{
45             int mid = l+r>>1;
46             return merge(query(l,mid,L,R,p<<1), query(mid+1,r,L
47                             ,R,p<<1|1));
48         }
49     };
50
51
52 int main(){
53     ios::sync_with_stdio(false);
54     cin.tie(0);
55     int n,Q;
56     cin>>n>>Q;
57     for(int i = 1;i<=n;i++){
58         cin>>da[i];
59     }
60     seg_tree st;
61     st.build(1,n,1);
62     int L,R;
63     for(int i = 1;i<=Q;i++){
64         cin>>L>>R;
65         S ans = st.query(1,n,L,R,1);
66         cout<<ans.mx - ans.mi<<endl;
67     }
68 }

```

6.8.3 思路

首先我们需要介绍一棵树的欧拉序列。

定义: 考虑一棵树的 dfs 遍历 (最后要回到根节点), 每遇到一个结点 (无论是否是回溯的) 便将序号其添加到序列中, 最后得到的序列便是欧拉序列。这里的序号等于将第一次遇到该结点的时刻排序后的次序。

引理 8 一棵树的欧拉序列长度为 $2n - 1$. (n 是结点个数。)

证明:

采取归纳法, 当 $n = 1$ 时, 欧拉序列长度为 1, 成立。

考虑一个子树 T_r 的根节点 r , 设其子树为 T_1, T_2, \dots, T_l ,

记函数 $f(T_r)$ 为子树欧拉序列长度, $c(T_r)$ 为子树的大小,

从 r 出发回到 r 过程中经过 r 的次数为 $l + 1$. 因此最后欧拉序列长度为

$$\begin{aligned} f(T_r) &= \sum_{i=1}^l f(T_i) + l + 1 \\ &= \sum_{i=1}^l (2c(T_i) - 1) + l + 1 \\ &= 2(c(T_r) - 1) - l + l + 1 \\ &= 2c(T_r) - 1 \end{aligned}$$

于是原命题得证。

引理 9 对于树上两点 u, v , u, v 的最近公共祖先是欧拉序列中 u, v 对应位置 (如果有多个, 任取一个) 组成的区间中的最小值。

证明: 首先 dfs 的过程中对于每一条边最多经历 2 次, 其次最近公共祖先必在上述区间中。

令 r 为满足上述性质的结点, 若 r 不为 u, v 的最近祖先, 则 $r- > \text{lca}(u, v)$ 将会被遍历至少 3 次 ($r- > u, u- > r, r- > \text{lca}(u, v)$)。

与 dfs 性质不符, 因此原命题证明。

6.8.4 例 17

洛谷-P3379 【模板】最近公共祖先 (LCA) 题面同例 11, 模板题, 使用 RMQ 和线段树求解版本。

Code:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define MAX_SIZE 500001
4 #define P_INF 1'000'000'000
5
6 int euler_map[MAX_SIZE];
```

```

7  int euler_map_inv[MAX_SIZE];
8  int euler_path[MAX_SIZE<<1];
9  int map_path_ind[MAX_SIZE];
10 bool vis[MAX_SIZE];
11
12 struct seg_tree
13 {
14     int s[MAX_SIZE<<3];
15     void build(int l,int r,int p){
16         if(l==r){
17             s[p] = euler_path[l];
18             return;
19         }
20         int mid = l+r>>1;
21         build(l,mid,p<<1);
22         build(mid+1,r,p<<1|1);
23         s[p] = min(s[p<<1],s[p<<1|1]);
24     }
25
26     int query(int l,int r,int L,int R,int p){
27         if(L<=l && R>=r) return s[p];
28         else if(l>R || r<L) return P_INF;
29         else{
30             int mid = l+r>>1;
31             return min(query(l,mid,L,R,p<<1),query(mid+1,r,L,R,
32                 p<<1|1));
33         }
34     };
35
36     seg_tree st;
37
38     int main(){
39         int n,m,root;
40         cin>>n>>m>>root;
41         vector adj(n+1,vector<int>(0));
42         for(int i = 1;i<=n-1;i++){
43             int u,v;
44             cin>>u>>v;

```

```

45         adj[u].push_back(v);
46         adj[v].push_back(u);
47     }
48     vis[root] = true;
49     euler_map[root] = 1;
50     euler_map_inv[1] = root;
51     int dfs_order = 1;
52     int ep = 0;
53     auto dfs_euler = [&](auto self,int root)->void{
54         vis[root] = true;
55         euler_path[++ep] = euler_map[root];
56         map_path_ind[euler_map[root]] = ep;
57         for(int v : adj[root]){
58             if(vis[v]==false){
59                 euler_map[v] = ++dfs_order;
60                 euler_map_inv[dfs_order] = v;
61                 self(self,v);
62                 euler_path[++ep] = euler_map[root];
63                 map_path_ind[euler_map[root]] = ep;
64             }
65         }
66     };
67     dfs_euler(dfs_euler,root);
68     st.build(1,2*n-1,1);
69     for(int i = 0;i<m;i++){
70         int u,v;
71         cin>>u>>v;
72         u = map_path_ind[euler_map[u]];
73         v = map_path_ind[euler_map[v]];
74         if(u>v) swap(u,v);
75         cout<<euler_map_inv[st.query(1,2*n-1,u,v,1)]<<endl;
76     }
77 }

```

7 树链剖分

7.1 概述

定义: 将整棵树剖分为若干条链状结构, 使用其余的数据结构维护信息。

树链剖分有很多种, 如重链剖分、长链剖分、实链剖分, 未明确说明的情况下本文的树链剖分都特指重链剖分。

树链剖分在以下的操作场景种很有用:

1. 更新结点 u 和 v 路径上的所有结点。
2. 查询结点 u 到 v 路径上的和、最大值、最小值等符合**结合律**的函数值。

7.2 数据结构定义

1. 重儿子: 儿子结点中子树最大的一个。
2. 轻儿子: 非重儿子的儿子。
3. 重边: 连接一个结点和其重儿子的边。
4. 轻边: 连接一个结点和其任意轻儿子的边。
5. 重链: 由重边组成的路径, 通常将只有没有重边相连的叶子结点也视为一条重链。
6. 重链头: 一条重链中深度最低的结点, 记 $\text{top}(u)$ 表示结点 u 所在的重链的重链头。
7. 轻链: 由轻边组成的路径。

7.3 性质

引理 10 对于任意结点 x 和其祖先结点 y , 从 $x \rightarrow y$ 的路径中经过轻边的个数不超过 $O(\log(n))$ 。

证明:

假设 $u \rightarrow v$ 是其经过的一条轻边, 其中 u 是 v 的父亲结点。

根据定义, u 必存在一重儿子 v' , 且 $\text{size}(T_{v'}) \geq \text{size}(T_v)$ 。

因此 $\text{size}(T_u) \geq 2\text{size}(T_v)$ 。

假设 l 表示 $u \rightarrow v$ 经过轻边的条数, 根据上述迭代过程, 则有

$$\begin{aligned} \text{size}(T_y) \geq 2^l \text{size}(T_x) &\Leftrightarrow l \leq \log(\text{size}(T_y)/\text{size}(T_x)) \\ &\Leftrightarrow l \leq O(\log(n)). \end{aligned}$$

引理 11 对于树上任意结点 x 和 y , 从 $x \rightarrow y$ 的路径中经过轻边的个数不超过 $O(\log(n))$.

证明:

假设 $r = \text{lca}(x, y)$, 因此 $x \rightarrow y = x \rightarrow r \rightarrow y$.

根据引理 10, $x \rightarrow r, r \rightarrow y$ 均为 $O(\log(n))$, 原命题得证。

7.4 树链剖分解决 lca 问题

7.4.1 思路

分析可得 lca 新的递推表达式:

不失一般性, 这里约定 $h(\text{top}(u)) \geq h(\text{top}(v))$, 可得 $\text{lca}(u, v)$ 在不同条件下的递推式:

$$\begin{cases} u, \text{top}(u) = \text{top}(v) \wedge h(u) \leq h(v) \\ v, \text{top}(u) = \text{top}(v) \wedge h(u) > h(v) \\ \text{lca}(\text{top}(\text{fa}(u)), v), \text{top}(u) \neq \text{top}(v) \end{cases} \quad (13)$$

$\text{top}(u)$ 的更新递推式如下:

$$\text{top}(u) = \begin{cases} \text{top}(\text{fa}(u)), & u \text{ 是重儿子,} \\ u, & u \text{ 是轻儿子.} \end{cases}$$

7.4.2 复杂度分析

预处理 top 等必要函数映射为 $O(n)$.

分析递推表达式, 执行 13 的次数等于 u, v 之间路径轻边的个数, 为 $O(\log(n))$.

因此单次查询 $\text{lca}(u, v)$ 复杂度为 $O(\log(n))$.

7.4.3 例 12

洛谷-P3379 【模板】最近公共祖先 (LCA)

题面同例 11, 模板题, 使用上述式子递推即可, 相较于倍增法, 使用树链剖分的常数较小。

代码:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int MAX_SIZE = 5e5+1;
5  int size_tr[MAX_SIZE];
6  int h[MAX_SIZE];
7  int top[MAX_SIZE];
8  int fa[MAX_SIZE];
9  int heavy_son[MAX_SIZE];
10 vector adj(MAX_SIZE, vector<int>(0));
11
12 void dfs_1(int root){
13     int sz = 1;
14     for(int v:adj[root]){
15         if(v==fa[root]) continue;
16         fa[v] = root;
17         h[v] = h[root]+1;
18         dfs_1(v);
19         sz+= size_tr[v];
20         if(heavy_son[root]==0 || size_tr[v]>size_tr[heavy_son[
21             root]]){
22             heavy_son[root] = v;
23         }
24     }
25     size_tr[root] = sz;
26 };
27 void dfs_2(int root, int tp){
28     for(int v:adj[root]){
29         if(v==fa[root]) continue;
30         if(v==heavy_son[root]){
31             top[v]= tp;

```

```

32         dfs_2(v, tp);
33     }
34     else{
35         top[v] = v;
36         dfs_2(v, v);
37     }
38 }
39 };
40 int lca(int u, int v){
41     while(top[u] != top[v]){
42         if(h[top[u]] < h[top[v]]) swap(u, v);
43         u = fa[top[u]];
44     }
45     return h[u] < h[v] ? u : v;
46 };
47 int main(){
48     h[0] = -1;
49     ios::sync_with_stdio(false);
50     cin.tie(0);
51     int n, m, root;
52     cin >> n >> m >> root;
53     int u, v;
54     for(int i = 0; i < n - 1; i++){
55         cin >> u >> v;
56         adj[u].push_back(v);
57         adj[v].push_back(u);
58     }
59     top[root] = root;
60     dfs_1(root);
61     dfs_2(root, root);
62     for(int i = 0; i < m; i++){
63         cin >> u >> v;
64         cout << lca(u, v) << "\n";
65     }
66 }

```

7.5 树链剖分与线段树

如果规定 dfs 每次必须先走重儿子, 那么每条重链的结点在 DFS 序中将是连续的。

那么对于每条重链上的信息, 我们可以使用线段树维护, 最终查询 $u \rightarrow v$ 的复杂度为 $O(n \log^2(n))$ 。

7.5.1 例 18

CSEC-Path Queries II

模板题, 只需要维护简单的路径求和。

注: 这题最后一个测试点 TLE 了, 经过长达几个小时的 DEBUG, 总结出以下特性:

1. endl 速度及其缓慢, 应该严格限制。
2. auto lambda 表达式对于运行速度的影响不大。
3. 关于左移位和右移位理论上会增加速度, 但是通常编译器可以优化, 因此影响不大。
4. scanf 速度慢于加了读入优化的 cin。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define MAX_SIZE 200'001
4  #define N_INF 0
5
6  int n;
7  int id_to_dfn[MAX_SIZE];
8  int dfn_to_id[MAX_SIZE];
9  int val[MAX_SIZE];
10 int dfs_order[MAX_SIZE];
11 int fa[MAX_SIZE];
12 int heavy_son[MAX_SIZE];
13 int size_tr[MAX_SIZE];
14 int top[MAX_SIZE];
15 int h[MAX_SIZE];
16

```

```

17 auto find_data(int i){
18     return val[dfn_to_id[i]];
19 }
20
21 struct seg_tree
22 {
23     int s[MAX_SIZE<<2];
24     void build(int l,int r,int p){
25         if(l==r){
26             s[p] = find_data(l);
27             return;
28         }
29         int mid = l+r>>1;
30         build(l,mid,p<<1);
31         build(mid+1,r,p<<1 | 1);
32         s[p] = max(s[p<<1], s[p<<1|1]);
33     }
34     int query(int l,int r,int L,int R,int p){
35         if(L<=l && R>=r) return s[p];
36         else if(l>R || r<L) return N_INF;
37         else{
38             int mid = l+r>>1;
39             return max(query(l,mid,L,R,p<<1), query(mid+1,r,L,R
40                 ,p<<1 | 1));
41         }
42     void update(int l,int r,int index,int val,int p){
43         if(l==r && l==index) s[p] = val;
44         else if(l>index || r<index) return;
45         else{
46             int mid = l+r>>1;
47             update(l,mid,index,val,p<<1);
48             update(mid+1,r,index,val,p<<1 | 1);
49             s[p] = max(s[p<<1], s[p<<1 | 1]);
50         }
51     }
52 };
53
54 seg_tree st;

```

```

55 int query_on_heavy_chain(int u,int v){
56     if(id_to_dfn[u]>id_to_dfn[v]) swap(u,v);
57     return st.query(1,n,id_to_dfn[u],id_to_dfn[v],1);
58 }
59
60 int merge_query(int u,int v){
61     int ans = 0;
62     while(top[u] !=top[v]){
63         if(h[top[u]] < h[top[v]]) swap(u,v);
64         ans = max(ans,query_on_heavy_chain(u,top[u]));
65         u = fa[top[u]];
66     }
67     ans = max(ans,query_on_heavy_chain(u,v));
68     return ans;
69 };
70 vector adj(0,vector<int>(0));
71 void dfs_1(int root){
72     size_tr[root] = 1;
73     for(int v:adj[root]){
74         if(v==fa[root]) continue;
75         fa[v] = root;
76         h[v] = h[root]+1;
77         dfs_1(v);
78         size_tr[root] += size_tr[v];
79         if(heavy_son[root]==0 || size_tr[heavy_son[root]]<
            size_tr[v]) heavy_son[root] = v;
80     }
81 };
82 int main(){
83     ios::sync_with_stdio(false);
84     cin.tie(0);
85     cout.tie(0);
86     // freopen("Code_18Test.txt","r",stdin);
87     // freopen("Code_18out.txt","w",stdout);
88     // clock_t begin_time = clock();
89     int q;
90     cin>>n>>q;
91     for(int i = 1;i<=n;i++){
92         cin>>val[i];

```

```

93     }
94     adj.resize(n+1);
95     for(int i = 1; i <= n-1; i++){
96         int u, v;
97         cin >> u >> v;
98         adj[u].push_back(v);
99         adj[v].push_back(u);
100    }
101    int root = 1;
102    h[root] = 0;
103    h[0] = -1;
104
105    dfs_1(root);
106    int dfs_or = 1;
107    dfs_order[root] = 1;
108    top[root] = root;
109    id_to_dfn[root] = 1;
110    dfn_to_id[1] = root;
111    auto dfn_update = [&](int node){
112        dfs_order[node] = ++dfs_or;
113        id_to_dfn[node] = dfs_order[node];
114        dfn_to_id[dfs_order[node]] = node;
115    };
116    auto dfs_2 = [&](auto self, int root) -> void{
117        if(heavy_son[root] != 0){
118            dfn_update(heavy_son[root]);
119            top[heavy_son[root]] = top[root];
120            self(self, heavy_son[root]);
121        }
122        for(int v: adj[root]){
123            if(v == fa[root] || v == heavy_son[root]) continue;
124            dfn_update(v);
125            top[v] = v;
126            self(self, v);
127        }
128    };
129    dfs_2(dfs_2, root);
130    st.build(1, n, 1);
131    for(int i = 1; i <= q; i++){

```

```
132     int type;
133     cin>>type;
134     if(type==1){
135         int u;
136         cin>>u>>val[u];
137         st.update(1,n,id_to_dfn[u],val[u],1);
138     }
139     else{
140         int u,v;
141         cin>>u>>v;
142         cout<<merge_query(u,v)<<"\n";
143     }
144 }
145 // cerr<<(clock()-begin_time)/1000.0<<"\n";
146 }
```

8 参考资料

- [1]. [OI-wiki](#)
- [2]. [Geeksforgeeks](#)
- [3]. [CodeForces-Tutorial](#)
- [4]. [CSDN 论坛](#)
- [5]. 算法导论 (第三版)
- [6]. [Wikipedia](#)
- [7]. [AC Library](#)

C++ 高级数据结构

ZeitHaum

2023 年 6 月 28 日

目录

1 greater

用于比较的函数对象,定义于 <functional>。类似的还有 less,greater_equal,less_equal。

使用时,需要先确保对应对象已经实现了相应的比较方法。**注意: 重载方法的参数一定是 const 函数, 参数变量也要被 const 修饰。**

例子:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 struct student{
5     int math_score;
6     int PE_score;
7     bool operator>(const student& s1)const{
8         if(this->math_score!=s1.math_score) return this->
            math_score > s1.math_score;
9         else return this->PE_score>s1.PE_score;
10    }
11 };
12
13
14 int main(){
15     vector<student> student_scores
16         {{80,100},{90,70},{80,99},{70,100}}; //初始化列表的嵌套
17         使用
18     greater<student> g;
19     auto print = [](student i){
20         cout<<i.math_score<<" "<<i.PE_score<<endl;
21     };
22     sort(student_scores.begin(),student_scores.end(),g);
23     for_each(student_scores.begin(),student_scores.end(),print)
24         ;
25 }
```

2 set

2.1 概述

set 中的元素都是唯一的，且以某种特定的顺序排列。set 中的元素不可以更改，但是可以插入或者移除，并且具有对数复杂度。set 常常使用红黑树实现。set 默认按升序排列（比较函数对象为 less<T>），使用 greater<T> 可以更改为降序排列。

set 和 vector 可以通过传入迭代器的方式互转，相关例子见??。

2.2 multiset

与 set 类似，区别在于允许元素重复。

2.3 algorithm

Algorithm 库中提供了求取 set 求交集、并集、差集和对称差集等函数。分别对应 set_intersection, set_union, set_difference, set_symmetric_difference。

例子：

```
1 class Solution {
2 public:
3     vector<int> intersection(vector<int>& nums1, vector<int>&
4         nums2) {
5         set<int> s1{nums1.begin(), nums1.end()};
6         set<int> s2{nums2.begin(), nums2.end()};
7         set<int> intersec;
8         set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end
9             (), inserter(intersec, intersec.begin()));
10        vector<int> ans{intersec.begin(), intersec.end()};
11        return ans;
12    }
13};
```

该算法也对 multiset 具备一样的效果，不同的是返回的集合中元素也会出现多次，和其运算的定义相同。如交集集合元素重复出现的次数等于在两个集合中出现的次数的较小值。

例子：

```

1  class Solution {
2  public:
3      vector<int> intersect(vector<int>& nums1, vector<int>&
        nums2) {
4          multiset<int> s1{nums1.begin(), nums1.end()};
5          multiset<int> s2{nums2.begin(), nums2.end()};
6          multiset<int> intersec;
7          set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end
            (), inserter(intersec, intersec.begin()));
8          vector<int> ans{intersec.begin(), intersec.end()};
9          return ans;
10     }
11 };

```

2.4 例题

2.4.1 力扣-丑数 II

链接:[力扣-265](#)

记丑数序列为 $U_n = \{u_1, u_2, \dots, u_n\}$, 且对于任意 $i < j, u_i < u_j$ 。显然, 对于任意丑数 u_i , 其可以写为 $u_i = 2^x 3^y 5^z (x, y, z \in \mathbb{N})$ 。于是对于任意丑数 u_i , 必然存在丑数 u_j , 使得 $u_j < u_i$ 且 $u_i = 2u_j$ 或者 $u_i = 3u_j$ 或者 $u_i = 5u_j$ 。

这样我们便可以维护一个待选集 S_n , 其元素为前 $n-1$ 个丑数的 2 倍数、3 倍数和 5 倍数。此时第 n 个丑数便是 S_n 的第 n 小数, 也即是 $S_n - U_{n-1}$ 的最小值。随着 n 变化, 我们可以通过动态删除 S_n 最小值和插入第 n 个丑数的 2 倍数、3 倍数、5 倍数来维护 $S_n - U_{n-1}$ 。此操作可以非常简单地通过 set 实现。

代码:

```

1  class Solution {
2  public:
3      int nthUglyNumber(int n) {
4          int cnt = 1;
5          set<long long> s;
6          s.insert(1ll);
7          while(true){
8              if(cnt==n) break;

```

```
9         long long minS = *s.begin();
10        s.insert(minS*2);
11        s.insert(minS*3);
12        s.insert(minS*5);
13        s.erase(s.begin());
14        cnt++;
15    }
16    return (int)(*s.begin());
17 }
18 };
```

3 优先队列

优先队列是类似于栈或者队列的抽象数据结构。每个优先队列中的元素均有一个优先级。与普通队列不同，高优先级的元素将比低优先级的元素先出队列。优先队列常常使用堆实现，但是也可以使用其他数据结构实现。

3.1 C++ 实现

C++ queue 库提供了插入和取出复杂度为 $\Theta(\log(n))$ 的优先队列，其中默认设置优先级为元素的大小，即元素大的将会被优先出队列。使用 `greater<T>` 可以得到另外一种对称的优先队列。C++ 中 `priority_queue` 使用堆实现，与自己写堆相比，封装的 `priority_queue` 可以避免因为操作失误导致堆失效。除了需要提供类模板外，`priority_queue` 还需要提供实现的数据类型，其必须具备 `front()`, `pop_back()` 和 `push_back()` 函数。C++ 标准库中 `vector` 和 `deque` 满足要求。

3.2 例题

3.2.1 力扣-数组中的第 k 大元素

链接:[力扣-215](#)。此题只需维护一个大小为 k 的优先队列即可，并设置将小元素的优先设置为最高即可。最终的答案便是优先队列的最顶端。

代码:

```
1 class Solution {
2 public:
```

```
3   int findKthLargest(vector<int>& nums, int k) {
4       int n = nums.size();
5       priority_queue<int, vector<int>, greater<int>> pq;
6       int i = 0;
7       for(; i < k; i++) pq.push(nums[i]);
8       for(; i < n; i++){
9           pq.push(nums[i]);
10          pq.pop();
11      }
12      return pq.top();
13  }
14  };
```

3.2.2 力扣-前 K 个高频元素

链接:[力扣-347](#)

上一题的进阶版，将优先级改为对应元素的频数即可。注意考虑此过程中和其他数据结构的配合使用。

```
1  class Solution {
2  public:
3      struct element{
4          int val;
5          int frequency;
6          bool operator>(const element& e) const{
7              if(this->frequency==e.frequency) return this->val>e
                  .val;
8              return this->frequency>e.frequency;
9          }
10         bool operator==(const element& e) const{
11             return this->frequency == e.frequency && this->val
                  == e.val;
12         }
13         bool operator<(const element& e) const{
14             if(this->frequency==e.frequency) return this->val<e
                  .val;
15             return this->frequency<e.frequency;
16         }
17     };
```

```
18
19     vector<int> topKFrequent(vector<int>& nums, int k) {
20         int n = nums.size();
21         map<int,int> freq;
22         for(int i = 0;i<n;i++){
23             if(freq.count(nums[i])==0) freq[nums[i]] = 1;
24             else freq[nums[i]]++;
25         }
26         set<element>s;
27         for(int i = 0;i<n;i++){
28             element e{nums[i],freq[nums[i]]};
29             s.insert(e);
30         }
31         priority_queue<element,vector<element>,greater<element>
32             >> pq;
33         auto p = s.begin();
34         for(int i = 0;i<k;i++,p++){
35             if(p==s.end()) break;
36             pq.push(*p);
37         }
38         while(p!=s.end()){
39             pq.push(*p);
40             pq.pop();
41             p++;
42         }
43         vector<int>ans(0);
44         while(!pq.empty()){
45             ans.push_back(pq.top().val);
46             pq.pop();
47         }
48         return ans;
49     };
```

4 双端链表

4.1 基本概述

包含在头文件 `<list>` 中，其迭代器是双向迭代器。只能 `++`，不能 `+n`。

4.2 迭代器不变特性

对于双向链表创建的元素，其对应的迭代器一经赋值直至销毁都不会改变。

见以下例子：

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main(){
5     list<int> a;
6     a.emplace_front(1);
7     auto a1 = a.begin(); // a1 -> 1
8     a.emplace_front(2);
9     a.emplace_back(3); // 链表结构: 2->1 -> 3
10    auto a2 = a.begin(); // a2 -> 2
11    auto a3 = a1++; // a3 -> 3
12    a.erase(a1); // 链表结构: 2 -> 3
13
14    //Check iterator Unchanged.
15    cout<<(a.begin()==a2);
16    auto new_a3 = ++a.begin();
17    cout<<(new_a3 == a3);
18
19 }
```

5 智能指针

5.1 概述

可以自动调用析构函数，避免内存泄露。

C++ 98 引入了 `auto_ptr`，C++ `unique_ptr`, `weak_ptr`, `shared_ptr`, `auto_ptr`

5.2 用法

见以下示例:

```
1 #include <memory> // 智能指针头文件
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 struct test{
6     int id;
7 };
8
9 int main(){
10     unique_ptr<int> int_ptr = unique_ptr<int>(new int()); // 构造函数
11     cout<<(*int_ptr); // 获取左值对象
12     unique_ptr<test> test_ptr = unique_ptr<test>(new test());
13     cout<<test_ptr->id;
14 }
```

、

注意给左值赋值智能指针、左值构造智能指针、new 智能指针都是未定义行为。

5.3 区别

unique_ptr 只允许单个指针对象管理这片内存 (互斥)。shared_ptr 支持多个指针对象管理, 并可以使用 use_count 方法查看对象数。weak_ptr 和 shared_ptr 共同作用, 解决内存锁死的情况, 一般不需要考虑。

6 tuple

6.1 基本概述

多元组。

6.2 基本用法

模板: tuple<Type t1, ...>;

已经重定义赋值类方法和偏序方法,但是偏序方法 (除了 ==) 在 C++20 被移除。

获取其第 i 个元素, `get<i>`(实例名)

6.3 示例代码

(力扣 15).

```
1 class Solution {
2 public:
3     unordered_map<int,int> hash;
4     //double find
5     unordered_map<int,int> cnt;
6     vector<int> nums_;
7     vector<vector<int>> threeSum(vector<int>& nums) {
8         //a number have more than 2 is useless(zero for 3).
9         for(int i = 0;i<nums.size();i++){
10             if(cnt.count(nums[i])==0){
11                 cnt[nums[i]] = 1;
12                 nums_.push_back(nums[i]);
13             }
14             else if(cnt[nums[i]]>=1 && cnt[nums[i]]<=2){
15                 cnt[nums[i]]++;
16                 nums_.push_back(nums[i]);
17             }
18             else if(cnt[nums[i]]==3 && nums[i]==0){
19                 cnt[nums[i]]++;
20                 nums_.push_back(nums[i]);
21             }
22         }
23         //build hash
24         for(int i = 0;i<nums_.size();i++){
25             hash[nums_[i]] = i;
26         }
27         set<tuple<int,int,int>> ret_set;
28         for(int i = 0;i<nums_.size();i++){
29             for(int j = i+1;j<nums_.size();j++){
30                 if(hash.count(-nums_[i] - nums_[j])!=0){
31                     int k = hash[-nums_[i] - nums_[j]];
```

```
32         if(k>j){
33             vector<int> vec = {nums_[i], nums_[j],
34                               nums_[k]};
35             sort(vec.begin(), vec.end());
36             ret_set.insert({vec[0],vec[1],vec[2]});
37         }
38     }
39 }
40
41 vector<vector<int>> ret;
42 for(tuple<int,int,int> x : ret_set){
43     ret.push_back({get<0>(x), get<1>(x), get<2>(x)});
44 }
45
46 return ret;
47 }
48 };
```

参考资料

- [1].[Cplusplus-set](#)
- [2].[CppReference-set](#)
- [3].[Wiki-PriorityQueue](#)
- [4].[CppReference-priority_queue](#)

C++ 进阶用法

ZeitHaum

2023 年 2 月 27 日

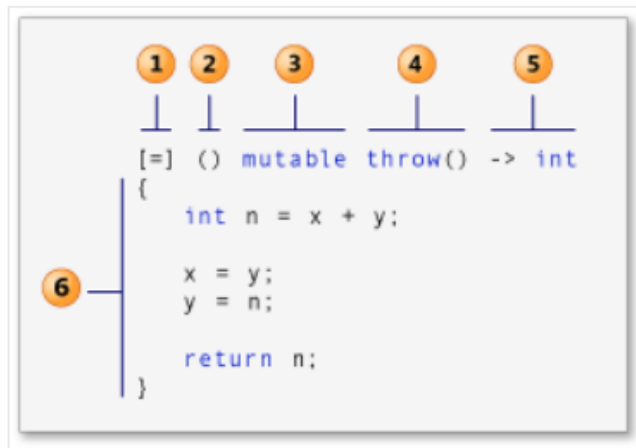
目录

1 C++ lambda 表达式	1
1.1 capture 子句	1
1.2 返回类型	2
1.3 mutable	2
2 迭代器	2
2.1 分类	2
2.2 不同容器中的迭代器	3
2.3 辅助函数	4
3 algorithm 库中常用函数	4
3.1 all_of	5
3.2 any_of	5
3.3 none_of	5
3.4 for_each	5
3.5 generate	6
3.6 generate_n	7
3.7 includes	7
3.8 inplace_merge	7
3.9 is_heap	8
3.10 is_heap_until	8
3.11 is_partitioned	8
参考文献	8

1 C++ lambda 表达式

C++ 11 以上特性。

下图显示了 lambda 语法的各个部分：



1. capture 子句（在 C++ 规范中也称为 Lambda 表达式捕获子句）。
2. 参数列表（可选）。（也称为 Lambda 声明符）
3. mutable 规范（可选）。
4. exception-specification（可选）。
5. trailing-return-type（可选）。
6. Lambda 体。

1.1 capture 子句

用于访问（捕获）外部变量。`[=]` 用于值捕获，`[&]` 用于引用捕获，`[this]` 捕获外部类指针 `this`（`[&]` 默认包含 `[this]`）。补充：`[args...]` 用于捕获外部可变

参数模板。

1.2 返回类型

编译器自动推导。也可以使用关键字“->”指定，此时不能省略空参数列表。

1.3 mutable

在按值捕获时无法在作用域内修改外部变量的值，使用 mutable 修饰后可以解决这个问题。但是修改仅限于 lambda 表达式内部生效。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main(){
5     int m = 1;
6     int n = 0;
7     [=] () mutable {
8         n++;
9         cout<<"The value of n is changed to "<<n<<". "<<endl;
10    }(); //Parentheses at the end indicate the default call.
11    cout<<"The value of n is "<<n<<". "<<endl;
12 }
```

输出结果为

```
The value of n is changed to 1.
The value of n is 0.
```

2 迭代器

迭代器用于访问顺序容器（主要是 vector 和数组）。

2.1 分类

分为正向迭代器、常量正向迭代器、反向迭代器、常量反向迭代器。比较常用的是正向迭代器和反向迭代器，反向迭代器的开始和结束分别为 rbegin() 和 rend()。

二者区别在于正向迭代器 ++ 返回顺序容器后一个数，后者返回前一个数。以下是二者使用的一个例子：

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main(){
5     vector<int> arr {1,2,3,4,5,6,7,8,9};
6     cout<<"The elements of the array are ";
7     for(vector<int>::iterator i = arr.begin();i<arr.end();i++){
8         cout<<*i<<" ";
9     }
10    cout<<"."<<endl;
11    cout<<"The elements of the reversed array are ";
12    for(vector<int>::reverse_iterator i = arr.rbegin();i<arr.
13        rend();i++){
14        cout<<*i<<" ";
15    }
16    cout<<"."<<endl;
17 }
```

输出结果为

```
The elements of the array are 1 2 3 4 5 6 7 8 9 .
The elements of the reversed array are 9 8 7 6 5 4 3 2 1 .
```

数组的迭代器就是指针，使用“数组名 +N”的形式表示。

2.2 不同容器中的迭代器

根据容器类型的不同，可将迭代器分为正向迭代器、双向迭代器、随机访问迭代器，限制依次呈递减趋势。其中数组和 vector 的迭代器都是随机访问迭代器。其支持的功能如下：

- $p+=i$: 使得 p 往后移动 i 个元素。
- $p-=i$: 使得 p 往前移动 i 个元素。
- $p+i$: 返回 p 后面第 i 个元素的迭代器。
- $p-i$: 返回 p 前面第 i 个元素的迭代器。
- $p[i]$: 返回 p 后面第 i 个元素的引用。

另外 $p_2 - p_1$ 和 $p_1 < p_2$ 均有定义 (和索引类似)。
不同容器的迭代器类型如下:

容器	迭代器功能
vector	随机访问
deque	随机访问
list	双向
set / multiset	双向
map / multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

2.3 辅助函数

C++ 中关于迭代器的辅助函数为 `advance`、`distance`、`iter_swap`; 其功能如下:

STL 中有用于操作迭代器的三个函数模板, 它们是:

- `advance(p, n)`: 使迭代器 `p` 向前或向后移动 `n` 个元素。
- `distance(p, q)`: 计算两个迭代器之间的距离, 即迭代器 `p` 经过多少次 `++` 操作后和迭代器 `q` 相等。如果调用时 `p` 已经指向 `q` 的后面, 则这个函数会陷入死循环。
- `iter_swap(p, q)`: 用于交换两个迭代器 `p`、`q` 指向的值。

3 algorithm 库中常用函数

`algorithm` 是 C++ 标准库之一, 需使用 `using namespace std;` 语句引入名称空间。

`algorithm` 库函数具有丰富的可扩展性, 这些需要使用 `lambda` 表达式和迭代器实现。

3.1 all_of

对列表中的元素执行谓词，如果都为真返回 true.

例子:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main(){
5     //Check the elements in the vector whether are all even
6     //numbers.
7     vector<int> arr {1,3,5,7,9,11};
8     vector<int> arr2 {2,4,6,8,10};
9
10    auto even_check = [](vector<int>& arr){
11        if(all_of(arr.begin(),arr.end(), [&](int i){return i
12            %2==1;})) cout<<"Check passed."<<endl;
13        else cout<<"Check unpassed."<<endl;
14    };
15    even_check(arr);
16    even_check(arr2);
17 }
```

输出:

```
Check passed.
Check unpassed.
```

3.2 any_of

和 all_of 区别是有任一为真即返回 true.

3.3 none_of

无一为真返回 true.

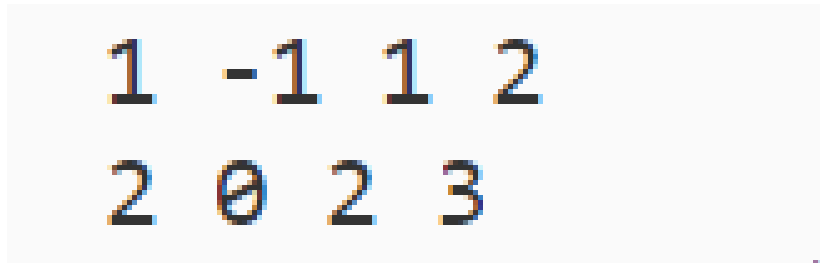
3.4 for_each

为每个函数执行操作，输入可以是函数指针也可以是 lambda 表达式。

例子:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define int long long
4
5 //function 1
6 void func1(long long &i){
7     i--;
8 }
9
10 signed main(){
11     auto func2 = [&](long long &i){
12         i++;
13     };
14     vector<int> arr{2,0,2,3};
15     auto print_arr = [&]() {for_each(arr.begin(),arr.end(),[](
16         int i){cout<<" "<<i;});});//Print the array.
17
18     for_each(arr.begin(),arr.end(),func1);
19     print_arr();
20     cout<<endl;
21     for_each(arr.begin(),arr.end(),func2);
22     print_arr();
23     cout<<endl;
24 }
```

输出结果为



3.5 generate

类似于 `for_each`, 只是更新方式由参数指针修改变为返回值赋值。

3.6 generate_n

类似于 generate, 只是结束迭代器被换为了大小 n。(从开始迭代器开始, 包含开始迭代器。)

3.7 includes

对于两个**已经排序好的 (增序)** 的范围 [first1,last1) 和 [first2,last2), 检查对于 [first2,last2) 中的元素, 是否**所有的**元素都被包含在 [first1,last1) 中。

若 [first1,last1) 为空, C++98 返回不确定值, 而 C++11 返回真。

例子:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main(){
5     int a[] = {1,2,3,4,5,6,7,8,9};
6     int b[] = {};
7     int c[] = {2,4,5,8};
8     int d[] = {7,8,9,10};
9     auto check = [&a](int* arr,int size){//Size is the size of
10         the array.
11         if (includes(a,a+9,arr,arr+size)) cout<<"All elements are
12             in the array."<<endl;
13         else cout<<"Some elements are not in the array."<<endl;
14     };
15     check(b,0);
16     check(c,4);
17     check(d,4);
18 }
```

输出结果为

```
All elements are in the array.
All elements are in the array.
Some elements are not in the array.
```

3.8 inplace_merge

合并两部分**已经排好序**的数组, 不常用。

3.9 `is_heap`

检查一个数组是否是一个堆。

3.10 `is_heap_until`

返回一个数组第一个不满足堆性质的非法字符。

3.11 `is_partitioned`

若满足谓词性质的元素均在不满足谓词的元素前返回 `true`, 否则返回 `false`.

参考文献

- [1]. [Microsoft C++、C 和汇编程序](#)
- [2]. [C++ algorithm 头文件下常用函数整理](#)
- [3]. [C++ 迭代器 \(STL 迭代器\) iterator 详解](#)
- [4]. [cplusplus.com-algorithm](#)

C++ 再温习

ZeitHaum

2023 年 2 月 20 日

目录

1 语法基础	1
1.1 include	1
1.2 main	1
1.3 标识符	1
1.4 语句	1
1.5 终止符	1
1.6 头文件名	2
1.7 名称空间	2
1.8 define	2
1.9 测试代码	3
2 输入输出	4
2.1 性能	4
2.2 占位符	5
2.3 测试代码	5
3 变量	6
3.1 内存空间	6
3.2 整数	7
3.3 bool	7
3.4 char	7
3.5 void	8
3.6 const	8
3.7 浮点数	8
3.7.1 基本概述	8
3.7.2 浮点常量	8
3.7.3 浮点数输出	8
3.8 类型转换	9
3.8.1 隐式类型转换	9
3.9 强制类型转换	9
3.10 auto	10
3.11 测试代码	10

4 运算	11
4.1 自增自减运算符	11
4.2 逗号运算符	11
4.3 赋值语句	11
4.4 运算优先级	11
4.5 结合性	11
4.5.1 a[i++] 和 a[++i]	12
4.6 除法的类型转换	12
4.7 测试代码	12
5 流程控制语句	13
5.1 if-else 控制语句	13
5.1.1 if	13
5.1.2 else	13
5.1.3 else if	13
5.2 逻辑运算	13
5.2.1 &&、 、!	13
5.2.2 细节	14
5.3 ctype 字符函数库	14
5.4 ?:	15
5.5 switch-case 语句	15
5.6 break 和 continue	16
5.7 测试代码	16
6 循环	17
6.1 for	17
6.1.1 基本结构	17
6.1.2 和逗号运算符使用的技巧	17
6.1.3 基于范围的循环	18
6.2 while	18
6.3 do-while	18
6.4 测试代码	18
7 C++ 高级数据类型	19

7.1	一维数组	19
7.1.1	基本概述	19
7.1.2	数组越界	19
7.1.3	数组初始化	19
7.1.4	C++11 的数组初始化	20
7.2	多维数组	20
7.3	字符串	20
7.3.1	C 风格字符串	20
7.3.2	C++ 风格字符串	21
7.3.3	字符串 I/O: 读取一行	21
7.3.4	其他形式的字符串	22
7.4	结构体	22
7.4.1	结构体初始化	23
7.4.2	结构数组	23
7.4.3	结构中的位字段	23
7.5	共用体	24
7.6	枚举	24
7.6.1	声明和赋值	24
7.6.2	指定枚举的值	24
7.7	测试代码	24
	参考资料	27

工欲善其事，必先利其器。

1 语法基础

1.1 include

将文件的内容复制到该处，include 引入的文件也称为头文件。**增加 include 的内容只会增加编译时间，对运行时间几乎没有影响。**

1.2 main

供系统或外部程序调用，return 0 表示运行成功，main 函数默认返回 return 0。

1.3 标识符

作为变量名的一组字符，具备以下特征：

1. 允许出现英文、数字、下划线。
2. 数字不能出现在第一个字符。
3. 不允许和 C++ 关键词同名。C++ 关键字可见于：[C++ 的关键字（保留字）完整介绍](#)

1.4 语句

需要执行的操作。

1.5 终止符

C++ 中即 “;”，其是语句的组成部分。

1.6 头文件名

表 2.1 头文件命名约定			
头文件类型	约定	示例	说明
C++旧式风格	以.h结尾	iostream.h	C++程序可以使用
C旧式风格	以.h结尾	math.h	C、C++程序可以使用
C++新式风格	没有扩展名	iostream	C++程序可以使用，使用 namespace std
转换后的 C	加上前缀 c，没有扩展名	cmath	C++程序可以使用，可以使用不是 C 的特性，如 namespace std

1.7 名称空间

避免重名。使用编译指令语句 using 导入。具备两种格式：

1. using namespace std;(全局)
2. using std::cin;(局部)

1.8 define

也称为宏定义语句，本质是文本替换。可以识别变量。例如：

```
1 #define pi 3.14159
2 #define sum(x,y) (x)+(y)
```

注意由于是文本替换，有可能引起运算顺序上的错误。如 2*sum(1,2) 使用上述文本替换后会错误地先计算乘法。

define 可以递归调用，如：

```
1 #define pi 3.1415926
2 #define calc_circle_area(x) pi*(x)*(x)
```

define 可以通过 # 函数将标识符转化为字符串，或者通过 ## 将两个标识符合并为新标识符。如：

```
1 #define tostring(a) (#a)
2 #define mergestring(a,b) (a##b)
```

调用第一个宏可以得到值为 a 的字面量，调用第二个宏可以得到标识符为 ab 的变量（需要提前定义，否则会报错。）

可以使用 #undef 结束宏定义，在 #undef 后的代码将无法使用对应的宏。

```
1 #undef pi
```

define 的另一大应用是条件编译。首先定义一个没有值的变量，如

```
1 #define DEBUG
```

在代码中可以通过 `#if defined(DEBUG)` 或 `#ifdef DEBUG` 语句对代码进行选择执行。其对应的取反操作为 `#if !defined(DEBUG)` 和 `#ifndef DEBUG`。表示其余情况用 `#else`，结束语句是 `#endif`。

1.9 测试代码

对本部分内容的测试代码：

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 #define pi 3.1415926
5 #define calc_circle_area(x) pi*(x)*(x)
6 #define toString(a) (#a)
7 #define mergestring(a,b) (a##b)
8 #define DEBUG
9
10 void testfunc(){
11     cout<<M_PI<<endl;
12     cout<<calc_circle_area(3.0)<<endl;
13     int x = 1;
14     int y = 2;
15     int xy = 3;
16     cout<<toString(x)<<endl;
17     cout<<mergestring(x,y)<<endl;
18     #undef pi
19     //cout<<pi<<endl; //未定义语句，将会报错。
20     #if defined(DEBUG)
21         cout<<"Enter DEBUG!"<<endl;
22     #else
23         cout<<"No DEBUG defined."<<endl;
24     #endif
25 }
```

```
26     #ifndef DEBUG
27         cout<<"Donot define DEBUG."<<endl;
28     #endif
29 }
30
31 int main(){
32     // cout<<M_PI;
33     testfunc();
34 }
```

2 输入输出

2.1 性能

std::cin 和 std::cout 相较于 C 语言的 scanf 和 printf 速度较慢。

使得 std::cin 速度较慢的原因之一为兼容 stdio 的开关,可使用 std::ios::sync_with_stdio(false); 关闭。此时 C++ 的输入输出和 C 语言输入输出解除绑定,此时不能同时使用 cin 和 scanf 或同时使用 cout 和 printf。

其次可以解除输入流和输出流的绑定。此时如果需要先 cout 再 cin 需要手动 flush。如

```
1 cin.tie(0);
2 cout<<"Hello,C++!";
3 int s;
4 cin>>s;
```

此时执行上述代码就可能会先输入 s 再输出字符串。注意:endl 会强制清空缓存,因此下述代码不会出现问题:

```
1 cin.tie(0);
2 cout<<"Hello,C++!"<<endl;
3 int t;
4 cin>>t;
```

其余更多优化 (getchar、putchar、fread、fwrite、mmap) 等参见[OI-wiki-OI 优化](#).

2.2 占位符

如下表 (注意:C++ 没有**%b** 输出二进制的方式。):

符号	含义	符号	含义
%d	十进制整数	%g	小数或者科学计数
%f	浮点数	%i	十进制、八进制、十六进制数
%s	字符串	%o	八进制数
%c	字符	%x	十六进制数
%p	指针	%%	%
%e	科学计数法	%u	无符号数
%[]	正则表达式读取字符集	%n	当前已有的字符数

%n 的用法举例如下:

```
1 int n;
2 scanf("%s%n",&str,&n);
3 cout<<toString(str)<<": "<<str<< " "<<toString(n)<<": "<<n<<"\n";
```

在控制台输入 “hello,world”, 终端输出:

```
str:hello,world n:11
```

%[] 是 C 语言的正则表达式用法, 当遇到第一个不满足 [] 内正则表达式的字符便会停止读取。注意需要导入 stdio 包 (不能同时导入 iostream 包)。一个简单的筛选数字的例子如下:

```
1 char x[80];
2 scanf("%[0-9]",x);
3 printf("%s",x);//注意:x已经是指针, 不能再次取地址。
```

输入 “123456789test”, 其输出只会读取前几个数字:

```
123456789test
123456789
```

2.3 测试代码

```
1 #include <iostream>
2 #include <stdio.h>
3 #define toString(a) #a
4 using namespace std;
5
6 void testfunc(){
7     ios::sync_with_stdio(false);
8     cin.tie(0);
9     cout<<"Hello,C++!";
10    int s;
11    cin>>s;
12    cout<<"Hello,C++!"<<endl;
13    int t;
14    cin>>t;
15    char str[200];
16    int n;
17    scanf("%s%n",&str,&n);
18    cout<<toString(str)<<":"<<str<<" "<<toString(n)<<":"<<n<<"\n";
19    // char x[80];
20    // scanf("%[0-9]",x);
21    // printf("%s",x);//注意:x已经是指针,不能再次取地址。
22 }
23
24 int main(){
25     testfunc();
26 }
```

3 变量

3.1 内存空间

不同类型的变量占据的内存空间不同。几个常见的 2 的幂次:

1. $2^{15} = 32768$

2. $2^{16} = 65536$

3. $2^{31} = 2147483648$

4. $2^{63} \approx 9.2 \times 10^{18}$

使用 `sizeof` 可以查看某个变量所占据的空间 (单位字节), 库 `climits` 可以查看各种类型可以表达的最大值和最小值。

3.2 整数

`short` 是 `short integer` 的简称, `long` 是 `long integer` 的简称。C++ 对不同整数的位数定义如下:

- `short` 至少 16 位;
- `int` 至少与 `short` 一样长;
- `long` 至少 32 位, 且至少与 `int` 一样长;
- `long long` 至少 64 位, 且至少与 `long` 一样长。

C++ 默认将证书字面量当作 `int` 类型。

整数类型的省略和顺序调整:

等价的类型表述

在不引发歧义的情况下, 允许省略部分修饰关键字, 或调整修饰关键字的顺序。这意味着同一类型会存在多种等价表述。

例如 `int`, `signed`, `int signed`, `signed int` 表示同一类型, 而 `unsigned long` 和 `unsigned long int` 表示同一类型。

3.3 bool

`bool` 类型占据 1 字节的空间。

3.4 char

占据 1 字节空间。ASCII 码中转义字符的定义如下:

字符名称	ASCII 符号	C++代码	十进制 ASCII 码	十六进制 ASCII 码
换行符	NL (LF)	\n	10	0xA
水平制表符	HT	\t	9	0x9
垂直制表符	VT	\v	11	0xB
退格	BS	\b	8	0x8
回车	CR	\r	13	0xD
振铃	BEL	\a	7	0x7
反斜杠	\	\\	92	0x5C
问号	?	\?	63	0x3F
单引号	'	\'	39	0x27
双引号	"	\"	34	0x22

3.5 void

表示无类型。

3.6 const

表示常量。

3.7 浮点数

3.7.1 基本概述

C++ 规定了三种浮点数:float、double、long double。在不同的机器中位数一般不一致,通常 float 有 32 位, double 64 位, long double 128 位。可使用 cfloat 库中的关键字查看。

3.7.2 浮点常量

浮点常量默认为 double 类型,如果需要声明为 float 或 long double,需要分别添加后缀 F 和 L(不区分大小写)。

3.7.3 浮点数输出

这里只介绍一两种通用的方法,更多的方法暂时省略。**1. 保留指定位数**使用 C 语言的格式化输出即可。

```
1 float pi = 3.1415926535;
2 printf("pi = %.6f\n",pi);
```

2. 输出末尾 0 使用 C 语言的格式化输出即可，注意 C 语言不支持 long double。

```
1 double pi2 = 3.15;
2 printf("pi = %.6f\n", pi2);
```

3. 舍去多余尾数如果是保留到整数位，可以使用 floor 函数或者 ceil 函数，如下例子：

```
1 float e = 2.718;
2 printf("%.0f\n", floor(e)); // 向下取整，需要引入 cmath 库
3 printf("%.0f\n", ceil(e)); // 向上取整
```

否则可以先左移再进行取整后再右移。

```
1 float test = 29.4525277;
2 printf("%.2f", floor(test*1e2)/1e2); // 保留两位小数(去尾法)
3 printf("%.2f", floor(test*1e2)/1e2); // 保留两位小数(进一法)
```

3.8 类型转换

3.8.1 隐式类型转换

为避免混乱，尽量避免进行隐性类型转换，尤其是无符号和有符号的转换。隐性类型转换的规则（需要特别留意 bool 和 char 的转换）：

- 先将 char, bool, short 等类型提升至 int（或 unsigned int，取决于原类型的符号性）类型；
- 若存在一个变量类型为 long double，会将另一变量转换为 long double 类型；
- 否则，若存在一个变量类型为 double，会将另一变量转换为 double 类型；
- 否则，若存在一个变量类型为 float，会将另一变量转换为 float 类型；
- 否则（即参与运算的两个变量均为整数类型）：
 - 若两个变量符号性一致，则将位宽较小的类型转换为位宽较大的类型；
 - 否则，若无符号变量的位宽不小于带符号变量的位宽，则将带符号数转换为无符号数对应的类型；
 - 否则，若带符号操作数的类型能表示无符号操作数类型的所有值，则将无符号操作数转换为带符号操作数对应的类型；
 - 否则，将带符号数转换为相对应的无符号类型。

3.9 强制类型转换

有两种方式，如 (long)a 和 long(a)。强制类型转换不会改变原本变量的值，而是创建一个新的变量用于参与表达式运算。

3.10 auto

auto 是 C++ 11 加入的特性，用于推测类别。注意 auto 不会进行类型转换，给予字面量时总是和字面量的默认类型一致。如 0 是 int, 0.0 是 double。

3.11 测试代码

```
1 #include<iostream>
2 #include<climits>
3 #include<cmath>
4 using namespace std;
5
6 void testfunc(){
7     bool b = false;
8     cout<<sizeof(b)<<endl;
9     cout<<LONG_LONG_MAX<<endl;
10    float pi = 3.1415926535;
11    printf("pi = %.6f\n",pi);
12    double pi2 = 3.15;
13    printf("pi = %.6f\n",pi2);
14    float e = 2.718;
15    printf("%.0f\n",floor(e));//向下取整，需要引入cmath库
16    printf("%.0f\n",ceil(e));//向上取整
17    float test = 29.4525277;
18    printf("%.2f\n",floor(test*1e2)/1e2);//保留两位小数(去尾法)
19    printf("%.2f\n",ceil(test*1e2)/1e2);//保留两位小数(进一法)
20 }
21
22 int main(){
23     testfunc();
24 }
```

4 运算

4.1 自增自减运算符

操作依次从左往右操作。如 `i++` 是先返回 `i` 再自增。尽量避免自增或自减运算符和其他运算符的嵌套，因为 C++ 未明确定义表达式中自增返回值的时机，不同的 C++ 编译器可能返回不同的结果。可参考 C++ Primerplus 中关于顺序点的讨论 (P134，第六版)。

4.2 逗号运算符

起分割运算符的作用，优先级最低，返回最右边的值。**不建议在普通情况下使用，可能引起误会。**

```
exp1, exp2, exp3; // 最后的值为 exp3 的运算结果。

Result = 1 + 2, 3 + 4, 5 + 6;
//得到 Result 的值为 3 而不是 11，因为赋值运算符 "="
//的优先级比逗号运算符高，先进行了赋值运算才进行逗号运算。

Result = (1 + 2, 3 + 4, 5 + 6);

// 若要让 Result 的值得到逗号运算的结果则应将整个表达式用括号提高优先级，此时
// Result 的值为 11。
```

4.3 赋值语句

赋值语句返回值是变量赋值之后的值，一般出现在误将“`==`”记为“`=`”的情况下。

4.4 运算优先级

总体上是!> 算术 > 关系 (如大于)> 与 > 或 > 赋值运算符。详细可参见[百度百科-运算优先级](#)。

4.5 结合性

先考虑运算优先级，再考虑结合性。结合性分为左结合和右结合，C 语言中单目运算符、条件运算符、赋值语句是右结合以外，其他基本均是左结合性。可参考帖子[C++ 运算符结合性（左结合，右结合）](#)

4.5.1 `a[i++]` 和 `a[++i]`

作为一个例子，背住即可。先计算 `i++`(或者 `[]` 内的东西)，再计算 `[]`。
例子：

```
1 int a[5] = {0,1,2,3,4};
2 int i = 0;
3 while(true){
4     if(i>=4) break;
5     cout<<a[++]<<endl;
6 }
7 int j = 0;
8 while(true){
9     if(j>=4) break;
10    cout<<a[j++]<<endl;
11 }
```

前者输出 1、2、3、4 后者输出 0、1、2、3。

4.6 除法的类型转换

除法的类型转换完全符合前文中的类型转换。两个整数需要浮点数结果需要先将其中一个数转换为浮点数。

4.7 测试代码

```
1 #include<iostream>
2 using namespace std;
3
4 void testfunc(){
5     int a[5] = {0,1,2,3,4};
6     int i = 0;
7     while(true){
8         if(i>=4) break;
9         cout<<a[++]<<endl;
10    }
11    int j = 0;
12    while(true){
13        if(j>=4) break;
```

```

14         cout<<a[j++]<<endl;
15     }
16 }
17
18 int main(){
19     testfunc();
20 }

```

5 流程控制语句

5.1 if-else 控制语句

5.1.1 if

功能介绍略。这里介绍一种避免将关系语句误写为赋值语句的方法：使用 `if(3==i)` 替换 `if(i==3)`。

条件运算符和错误防范

许多程序员将更直观的表达式 `variable == value` 反转为 `value == variable`，以此来捕获将相等运算符误写为赋值运算符的错误。例如，下述条件有效，可以正常工作：

```
if (3 == myNumber)
```

但如果错误地使用下面的条件，编译器将生成错误消息，因为它以为程序员试图将一个值赋给一个字面值（3总是等于3，而不能将另一个值赋给它）：

```
if (3 = myNumber)
```

假设犯了类似的错误，但使用的是前一种表示方法：

```
if (myNumber = 3)
```

编译器将只是把3赋给myNumber，而if中的语句块将包含非常常见的、而又非常难以发现的错误（然而，很多编译器会发出警告，因此注意警告是明智的）。一般来说，编写让编译器能够发现错误的代码，比找出导致难以理解的原因要容易得多。

5.1.2 else

略。

5.1.3 else if

略。

5.2 逻辑运算

5.2.1 &&、||、!

略。

5.2.2 细节

1. 使用 `a>3 && a<5` 而不是 `3<a<5`。后者会被视为 `(3<a)<5`。2. 符号! 的优先级高于关系运算符，因此取反要括起来。比如 `!x>1` 表示判断!x 和 1 的大小关系。3. C++ 支持使用 `and`、`or`、`not` 替换 `&&`、`||`、`!`。例如：

```
1 int x = 3;
2 int y = 2;
3 if(3==x and y>1 and y<3) cout<<"If test PASS!"<<endl;
```

5.3 ctype 字符函数库

提供判断字符 (char) 类型的很多函数，比用 if-else 更方便。如可以使用 `isalpha()` 判断是否为字母。以下是一个例子：

```
1 string a_str = "Hello,world!";
2 for(int i = 0;i<a_str.size();i++){
3     if(isalpha(a_str[i]))cout<<a_str[i];
4 }
5 cout<<endl;
```

程序的输出结果为 Helloworld。ctype 其他函数如下：

表 6.4 ctype 中的字符函数

函数名称	返回值
<code>isalnum()</code>	如果参数是字母数字，即字母或数字，该函数返回 true
<code>isalpha()</code>	如果参数是字母，该函数返回 true
<code>isctrl()</code>	如果参数是控制字符，该函数返回 true
<code>isdigit()</code>	如果参数是数字 (0~9)，该函数返回 true
<code>isgraph()</code>	如果参数是除空格之外的打印字符，该函数返回 true
<code>islower()</code>	如果参数是小写字母，该函数返回 true
<code>isprint()</code>	如果参数是打印字符 (包括空格)，该函数返回 true
<code>ispunct()</code>	如果参数是标点符号，该函数返回 true
<code>isspace()</code>	如果参数是标准空白字符，如空格、进纸、换行符、回车、水平制表符或者垂直制表符，该函数返回 true
<code>isupper()</code>	如果参数是大写字母，该函数返回 true
<code>isxdigit()</code>	如果参数是十六进制数字，即 0~9、a~f 或 A~F，该函数返回 true
<code>tolower()</code>	如果参数是大写字符，则返回其小写，否则返回该参数
<code>toupper()</code>	如果参数是小写字符，则返回其大写，否则返回该参数

5.4 ?:

?: 是 C++ 唯一的一个三目运算符。注意嵌套问题，当判断条件过多时推荐使用 if-else 语句。

5.5 switch-case 语句

switch 处填入一个表达式 (值必须在整数集合内)，case 语句填入一个整数值。根据表达式的值，程序跳转到对应的语句**依次**执行代码。要避免依次执行请使用 break 语句。如以下例子：

```
1  int z = 0;
2  while (1)
3  {
4      if(z>=4) break;
5      switch (z)
6      {
7          case 0:
8              cout<<"z is 0."<<endl;
9              break;
10         case 1:
11             cout<<"z is 1."<<endl;
12             break;
13         case 2:
14             cout<<"z is 2."<<endl;
15             break;
16         default:
17             cout<<"Error:z has a wrong value."<<endl;
18             break;
19     }
20     z++;
21 }
```

输出结果：

```
z is 0.
z is 1.
z is 2.
Error:z has a wrong value.
```


5.6 break 和 continue

略。

5.7 测试代码

```
1  #include<iostream>
2  #include<string>
3  #include<cctype>
4  using namespace std;
5
6  void testfunc(){
7      int x = 3;
8      int y = 2;
9      if(3==x and y>1 and y<3) cout<<"If test PASS!"<<endl;
10     string a_str = "Hello,world!";
11     for(int i = 0;i<a_str.size();i++){
12         if(isalpha(a_str[i])) cout<<a_str[i];
13     }
14     cout<<endl;
15     int z = 0;
16     while (1)
17     {
18         if(z>=4) break;
19         switch (z)
20         {
21             case 0:
22                 cout<<"z is 0."<<endl;
23                 break;
24             case 1:
25                 cout<<"z is 1."<<endl;
26                 break;
27             case 2:
28                 cout<<"z is 2."<<endl;
29                 break;
30             default:
31                 cout<<"Error:z has a wrong value."<<endl;
32                 break;
33     }
```

```
34         z++;
35     }
36 }
37
38 int main(){
39     testfunc();
40 }
```

6 循环

6.1 for

6.1.1 基本结构

for 循环基本结构:

```
for (初始化; 判断条件; 更新) {
    循环体;
}
```

任何一个部分都可以省略 (; 不能省略), 省略判断条件默认为真。

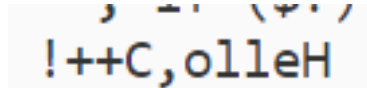
for 循环是入口条件循环, 每轮循环开始前都将计算循环条件, 因此当循环条件较为复杂时会降低效率 (视编译器而定)。

6.1.2 和逗号运算符使用的技巧

for 循环的另一个技巧是和逗号运算符的使用, 参见以下反转字符串的方法:

```
1 string s = "Hello,C++!";
2 int i,j;//必须提前声明, int j = s.size()-1不是一个表达式语句。
3 for(i = 0,j = s.size()-1;i<j;i++,j--){
4     char c = s[i];
5     s[i] = s[j];
6     s[j] = c;
7 }
8 cout<<s<<endl;
```

输出:



6.1.3 基于范围的循环

C++ 引入的新特性，类似于 Python 的范围遍历，**通常和 auto 结合使用**。一个简单的例子如下，用于输出数组全部元素。

```
1 float f[4] = {0.1,0.2,0.3,1.1};
2 for(auto x:f){
3     cout<<x<<endl;
4 }
```

6.2 while

功能略。永远循环的两种等价写法:

1. for(;;)
2. while(1)

6.3 do-while

先执行语句再判断条件，一般不常用。

6.4 测试代码

```
1 #include<iostream>
2 #include<string>
3 using namespace std;
4
5 void testfunc(){
6     string s = "Hello,C++!";
7     int i,j;//必须提前声明, int j = s.size()-1不是一个表达式语
8     句。
9     for(i = 0,j = s.size()-1;i<j;i++,j--){
10         char c = s[i];
```

```
10     s[i] = s[j];
11     s[j] = c;
12 }
13 cout<<s<<endl;
14 float f[4] = {0.1,0.2,0.3,1.1};
15 for(auto x:f){
16     cout<<x<<endl;
17 }
18 }
19
20 int main(){
21     testfunc();
22 }
```

7 C++ 高级数据类型

7.1 一维数组

7.1.1 基本概述

只能存访相同类型，声明长度必须是常量（被 `const` 修饰）。

7.1.2 数组越界

会触发段错误或者修改意外之外的值。

7.1.3 数组初始化

直接阅读以下代码和注释即可：

```
1 //初始化数组
2 int arr1[4];
3 int arr2[3] = {1,2,3};
4 int arr3[7] = {1,2,3}; //不满默认补零。
5 int arr4[10] = {0}; //快速初始化一个全为0的数组
6 int arr5[] = {12,23,445,464,1225}; //自动推测数组长度
```

打印数组的结果如下，可用于检验数组生成结果：

```
[0,0,0,0]
[1,2,3]
[1,2,3,0,0,0,0]
[0,0,0,0,0,0,0,0,0,0]
[12,23,445,464,1225]
```

注意: 在程序的非静态区声明的数组取值不一定为默认值 (0)。

7.1.4 C++11 的数组初始化

可以省略 “=” 号, 若 “{}” 中无内容将会视为全 0. 如以下代码:

```
1 int arr6[] {1,3,5,7,9};
2 int arr7[5] {};
```

7.2 多维数组

多维数组实质上在计算机内仍是按一维数组存储, 不过其下标进行了对应的换算。数组存储的方法有很多种, 如二维数组有按行优先和按列优先两种存储方法。实际中常常尽量选择局部性良好的方法。

另外, 多维数组应尽量使其存储位置连续, 这样可以利用 Cache 加速实现性能提升。C++ 中 vector 速度慢于自带的多维数组的原因便是 vector 行间数据存储不连续。详情可见[C++ 多维 vector 为什么比多维数组慢](#)。

7.3 字符串

7.3.1 C 风格字符串

一个包含 ‘\0’ 的 char 数组, 且 ‘\0’ 视为字符串的结束, 该字符以后的字符被忽略。

1. 初始化

可以使用数组或字符串的形式初始化, 注意数组的空间要给够。以下是例子:

```
1 //字符串
2 char str1[15] = {'H','e','l','l','o',' ',' ','w','o','r','l','d',' ',' ','\0'}; //末尾'\0'不可省略。
```

```
3 char str2[15] = "Hello,world!"; //多余空间均被赋值为'\0'。  
4 char str3[] = "Hello,world!"; //自动推测
```

2. 拼接字符串常量

C++ 允许以下操作:

```
1 cout<<"Hello"  
2 ",world!"<<endl;
```

当字符串很长时此技巧很有用。

3.cstring 库

cstring 库 (或 string.h) 提供了一些与 C 风格相关的函数, 如 strlen() 和 strcmp 函数。

```
1 printf("The size of str1 is %d.\n",strlen(str1));  
2 if(strcmp(str1,str2)) printf("%s","str1 = str2.\n");  
3 else printf("%s","str1!=str2.\n");
```

注意判断字符串相等不能使用"==", 这里"str1==str2" 返回的是两者地址是否相同, 如下:

```
1 printf("The size of str1 is %d.\n",strlen(str1));  
2 if(str1==str2) printf("%s","The address of str1 is equal to  
   str2.\n");  
3 else printf("%s","The address of str1 is not equal to str2.\n")  
   ;
```

7.3.2 C++ 风格字符串

使用的是 string 库, 具体用法参照官方文档。

7.3.3 字符串 I/O: 读取一行

对于 C 风格字符串, 有 cin.get(str,length) 和 cin.getline(str,lenth) 两个函数。这里 length 表示读取的字符串最大长度。当读取到换行符或者读取字符超过最大长度限制将会停止读取。对于 C++ 风格字符串, 有 getline(cin,str) 一个函数。示例如下:

```
1 char str5[50];  
2 char str6[50];
```

```

3 char test_c;
4 string str7;
5 cout<<"Readline begin."<<endl;
6 cin.getline(str5,20);
7 cin.get(str6,20);
8 //get会保留换行符，因此c必为换行符；对于cin可以忽略。
9 scanf("%c",&test_c);
10 cout<<int(test_c)<<endl;
11 getline(cin,str7);
12 cout<<"Readline end."<<endl;

```

输出可见 test_c 的 ASCII 编号为 10，正是代表换行符。

```

Readline begin.
test1
test2
10
test3
Readline end.

```

7.3.4 其他形式的字符串

如下表:

类型名称	别称	适用版本	占用比特数	字面量前缀	备注
wchar_t	宽字符类型	C++	16	L	用于国际化
char16_t	-	C++11	16	u	-
char32_t	-	C++11	32	U	-

另外 C++11 还可以使用前缀 “u8” 指定编码格式，“R” 表示原始字符类型。

7.4 结构体

C 语言不能省略 struct.

7.4.1 结构体初始化

如下代码，解析见注释：

```
1 //结构体声明
2 struct test
3 {
4     int x;
5     int y;
6     double d;
7     string s;
8 };
9 //采用数组的方式初始化
10 test t1 = {1,2,1.2,"test"}; //不省略等号
11 test t2 {1,2,1.3,"test2"}; //省略等号(C++11)
12 cout<<t1.d<<endl;
13 cout<<t2.s<<endl;
14 //声明结构体的同时初始化变量
15 struct position
16 {
17     float x;
18     float y;
19 } p1,p2;
20 cout<< p1.x-p2.x <<endl;
21
22 //省略结构体名,以后无法使用结构体名新建变量
23 struct {
24     double x;
25     double y;
26     string name;
27 } shop = {1.0,1.0,"Cake"};
28 cout<<shop.name<<endl;
```

7.4.2 结构数组

略。

7.4.3 结构中的位字段

使用“:”指定占据特定位数的结构成员，常用于低级编程。此处不展开。

7.5 共用体

只能同时存储多种数据类型中的一种，当一个变量拥有多种不同的数据类型时可以使用。常用于嵌入式编程节省内存，普通系统不常用，此处不展开。

7.6 枚举

7.6.1 声明和赋值

一种创建符号常量的方法，声明与 struct 类似。默认将枚举赋值为 0,1,2,...，枚举类型能隐式转换为整数，但是整数不可以隐式转换为枚举类型。枚举类型只定义了赋值语句，其他操作未定义。当整数超出了枚举类型范围也是未定义行为，在某些编译器中将会报错。例子如下：

```
1 enum color {red,green,white,black};
2 color c = red;
3 cout<<c<<endl;
4 //color c2 = 3; Error!
5 color c2 = (color)3;//OK
```

7.6.2 指定枚举的值

枚举类型允许有相同的值，枚举有效范围为枚举值中最大的值。C++ 允许枚举被赋值为 long long 类型，不允许非整数类型。例子如下：

```
1 enum bits {a = 2,one = 2,two = 4,three = 8,four = 16,super_long
    = 2147483649L};//可声明为 long long.
2 bits b = (bits)6;//Valid, In range.
3 cout<<bits::super_long<<endl;//访问枚举值使用::符号
```

7.7 测试代码

```
1 #include<iostream>
2 #include<cstring>
3 #include<string>
4 using namespace std;
5
```

```
6 //打印数组
7 void printfArray(int* arr_begin, int length){
8     cout<<"[";
9     for(int i = 0;i<length;i++){
10         if(i==0) printf("%d",arr_begin[i]);
11         else printf(",%d",arr_begin[i]);
12     }
13     cout<<""]"<<endl;
14 }
15
16 void testfunc(){
17     //初始化数组
18     int arr1[4]; //在非静态区声明不一定全0。
19     int arr2[3] = {1,2,3};
20     int arr3[7] = {1,2,3}; //不满默认补零。
21     int arr4[10] = {0}; //快速初始化一个全为0的数组
22     int arr5[] = {12,23,445,464,1225}; //自动推测数组长度
23     printfArray(arr1,4);
24     printfArray(arr2,3);
25     printfArray(arr3,7);
26     printfArray(arr4,10);
27     printfArray(arr5,5);
28     //C++11 数组初始化
29     int arr6[] {1,3,5,7,9};
30     int arr7[5] {};
31     printfArray(arr6,5);
32     printfArray(arr7,5);
33
34     //字符串
35     char str1[20] = {'H','e','l','l','o',' ',' ','w','o','r','l','d',' ','!',' ','\0','a','b'}; //末尾'\0'不可省略。
36     char str2[20] = "Hello,world!"; //多余空间均被赋值为'/0'。
37     char str3[] = "Hello,world!"; //自动推测
38     printf("str1:%s\n",str1);
39     printf("str2:%s\n",str2);
40     printf("str3:%s\n",str3);
41
42     cout<<"Hello"
43     ",world!"<<endl;
```

```
44
45     printf("The size of str1 is %d.\n",strlen(str1));
46     if(strcmp(str1,str2)) printf("%s","str1 = str2.\n");
47     else printf("%s","str1!=str2.\n");
48
49     printf("The size of str1 is %d.\n",strlen(str1));
50     if(str1==str2) printf("%s","The address of str1 is equal to
        str2.\n");
51     else printf("%s","The address of str1 is not equal to str2
        .\n");
52
53     char str5[50];
54     char str6[50];
55     char test_c;
56     string str7;
57     cout<<"Readline begin."<<endl;
58     cin.getline(str5,20);
59     cin.get(str6,20);
60     //get会保留换行符，因此c必为换行符；对于cin可以忽略。
61     scanf("%c",&test_c);
62     cout<<int(test_c)<<endl;
63     getline(cin,str7);
64     cout<<"Readline end."<<endl;
65
66     //结构体声明
67     struct test
68     {
69         int x;
70         int y;
71         double d;
72         string s;
73     };
74     //采用数组的方式初始化
75     test t1 = {1,2,1.2,"test"}; //不省略等号
76     test t2 {1,2,1.3,"test2"}; //省略等号(C++11)
77     cout<<t1.d<<endl;
78     cout<<t2.s<<endl;
79     //声明结构体的同时初始化变量
80     struct position
```

```
81     {
82         float x;
83         float y;
84     } p1,p2;
85     cout<< p1.x-p2.x <<endl;
86
87     //省略结构体名,以后无法使用结构体名新建变量
88     struct {
89         double x;
90         double y;
91         string name;
92     } shop = {1.0,1.0,"Cake"};
93     cout<<shop.name<<endl;
94
95     enum color {red,green,white,black};
96     color c = red;
97     cout<<c<<endl;
98     //color c2 = 3; Error!
99     color c2 = (color)3;//OK
100
101     enum bits {a = 2,one = 2,two = 4,three = 8,four = 16,
102               super_long = 2147483649L};//可声明为long long.
103     bits b = (bits)6;//Valid, In range.
104     cout<<bits::super_long<<endl;//访问枚举值使用::符号
105 }
106
107 int main(){
108     testfunc();
109 }
```

参考资料

- [1]. [OI-wiki](#).
- [2]. C++ Primerplus 中文版 (第六版).
- [3]. [菜鸟教程](#)
- [4]. [CSDN](#).

并查集

ZeitHaum

2023 年 4 月 1 日

目录

1 概述	1
2 如何查找?	1
3 如何合并?	1
4 性能优化	1
4.1 按尺寸合并	1
4.2 按秩合并	1
4.3 路径压缩	1
4.4 复杂度	1
4.5 应用	2
4.6 实现	2
5 相关例题	3
5.1 例题 1	3
5.1.1 Solution	3
5.1.2 Code	3
5.2 例题 2	7
5.2.1 Solution	7
5.2.2 Code	7
6 参考资料	9

1 概述

并查集是一种存储非重叠和不相交元素集合的数据结构。其支持查找和合并两种基本操作。

2 如何查找？

对于每个并查集，选择一个元素作为其**代表元素**。选择代表元素的方式有很多，一种基本方式是选择最大并查集中或者最小的元素。另外规定一个并查集的代表元素为其本身，且此元素必须属于并查集。

查找两个元素是否属于一个并查集只需检查其代表元素是否一致即可。

3 如何合并？

先找到两个并查集的代表元素，然后修改其中一个元素的代表元素为另外一个即可。

4 性能优化

4.1 按尺寸合并

合并时将大小小的并查集合并到尺寸大的并查集上。

4.2 按秩合并

定义每个并查集的秩为其树状结构的高度，合并时将低秩的并查集合并到高秩并查集是更优的选择。

4.3 路径压缩

每次查询时将查询节点直接和代表元素相连，将降低重复查询相同元素的复杂度。

4.4 复杂度

1. 若没有任何性能优化，单次查询和合并复杂度为 $O(n)$.

2. 若只带按秩合并或按尺寸合并优化, 对于 n 个元素和 m 次操作, 其复杂度为 $O(m \log(n))$.
3. 若只带路径压缩优化, 对于 n 次合并和 f 次查询操作, 其复杂度为 $O(n + f \cdot \log_{2+f/n}(n))$.
4. 若既有按秩合并和路径压缩, 对于 m 次操作和 n 次合并, 其复杂度为 $O(m\alpha(n))$. 其中 $\alpha(n)$ 为反 Ackerman 函数, 对于物理世界中任意可以写下的数, $\alpha(n)$ 不会超过 4.

4.5 应用

1. 最小生成树算法 (Kruskal).
2. 图论判环.
3.

4.6 实现

可用负数标记每个并查集的代表元素, 为了高效利用空间, 可令并查集代表元素的标记为并查集大小或者秩的相反数。由此根据按尺寸和按秩的不同有两种实现:

```

1  #include<bits/stdc++.h>
2  using namespace std;
3
4  struct DSU_size{
5      vector<int> e;
6      DSU_size(int N){
7          e = vector<int>(N,-1);
8      }
9      int find(int x){return e[x]<0?x:e[x] = find(e[x]);}
10     int size(int x){return -e[find(x)];}
11     bool check_same(int x,int y){return find(x)==find(y);}
12     bool unite(int x,int y){
13         x = find(x),y = find(y);
14         if(x==y) return false;
15         if(e[x]<e[y]) swap(x,y);//小的往大的合入。
16         e[y] += e[x];//注意更新顺序。

```



```

17         e[x] = y;
18         return true;
19     }
20 };
21
22 struct DSU_rank{
23     vector<int>e;
24     DSU_rank(int N){e = vector<int>(N,0);}
25     int find(int x){return e[x]<=0?x:e[x] = find(e[x]);}
26     int rank(int x){return -e[find(x)];}
27     bool check_same(int x,int y){return find(x)==find(y);}
28     bool unite(int x,int y){
29         x = find(x),y = find(y);
30         if(x==y) return false;
31         if(e[x]<e[y]) swap(x,y);
32         e[y] += e[x]==e[y]?0:-1;
33         e[x] = y;
34         return true;
35     }
36 };

```

5 相关例题

5.1 例题 1

[PTA 天体赛-练习集:L2-024 部落](#)

5.1.1 Solution

该题重点是 DSU 个数的求取，观察到每成功合并一次总的 DSU 个数将会减一，可记录总的减少值最后加上数组长度即是答案。

5.1.2 Code

按尺寸合并版本：

```

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

```

```

3
4 struct DSU_size{
5     vector<int> e;
6     DSU_size(int N){
7         e = vector<int>(N,-1);
8     }
9     int find(int x){return e[x]<0?x:e[x] = find(e[x]);}
10    int size(int x){return -e[find(x)];}
11    bool check_same(int x,int y){return find(x)==find(y);}
12    bool unite(int x,int y){
13        x = find(x),y = find(y);
14        if(x==y) return false;
15        if(e[x]<e[y]) swap(x,y); //小的往大的合入。
16        e[y] += e[x];
17        e[x] = y;
18        return true;
19    }
20 };
21
22
23 //带按尺寸合并和路径压缩
24 void solve(){
25     int n;
26     int MAX_SIZE = 10001;
27     DSU_size dsu = DSU_size(MAX_SIZE);
28     int actual_size = 0;
29     cin>>n;
30     int merge_cnt = 0;
31     for(int i = 0;i<n;i++){
32         int K;
33         cin>>K;
34         int rep;
35         cin>>rep;
36         actual_size = max(rep,actual_size);
37         int temp;
38         for(int j = 1;j<K;j++){
39             cin>>temp;
40             actual_size = max(temp,actual_size);
41             if(dsu.unite(temp,rep)) merge_cnt--;

```

```

42     }
43 }
44 int Q;
45 cin>>Q;
46 int x,y;
47 cout<<actual_size<<" "<<merge_cnt+actual_size<<endl;
48 for(int i = 0;i<Q;i++){
49     cin>>x>>y;
50     if(dsu.check_same(x,y)) cout<<"Y"<<endl;
51     else cout<<"N"<<endl;
52 }
53 }
54
55
56 int main(){
57     solve();
58 }

```

按秩合并版本:

```

1  #include<bits/stdc++.h>
2  using namespace std;
3
4  struct DSU_rank{
5      vector<int>e;
6      DSU_rank(int N){e = vector<int>(N,0);}
7      int find(int x){return e[x]<=0?x:e[x] = find(e[x]);}
8      int rank(int x){return -e[find(x)];}
9      bool check_same(int x,int y){return find(x)==find(y);}
10     bool unite(int x,int y){
11         x = find(x),y = find(y);
12         if(x==y) return false;
13         if(e[x]<e[y]) swap(x,y);
14         e[y] += e[x]==e[y]?0:-1;
15         e[x] = y;
16         return true;
17     }
18 };
19
20

```

```
21 //带按秩合并和路径压缩
22 void solve(){
23     int n;
24     int MAX_SIZE = 10001;
25     DSU_rank dsu = DSU_rank(MAX_SIZE);
26     int actual_size = 0;
27     cin>>n;
28     int merge_cnt = 0;
29     for(int i = 0;i<n;i++){
30         int K;
31         cin>>K;
32         int rep;
33         cin>>rep;
34         actual_size = max(rep,actual_size);
35         int temp;
36         for(int j = 1;j<K;j++){
37             cin>>temp;
38             actual_size = max(temp,actual_size);
39             if(dsu.unite(temp,rep)) merge_cnt--;
40         }
41     }
42     int Q;
43     cin>>Q;
44     int x,y;
45     cout<<actual_size<<" " <<merge_cnt+actual_size<<endl;
46     for(int i = 0;i<Q;i++){
47         cin>>x>>y;
48         if(dsu.check_same(x,y)) cout<<"Y"<<endl;
49         else cout<<"N"<<endl;
50     }
51 }
52
53 int main(){
54     solve();
55 }
```

5.2 例题 2

力扣-岛屿数量

5.2.1 Solution

本题关键点在于并查集对象不是 `int` 类型而是一个 `int` 对。可使用 `Std` 标准库的 `map` 替换 `vector`, 注意需要重载的操作符对象以及拷贝构造函数的写法标准, 诀窍在于参数中尽量考虑是否能添加 `&` 和 `const` 关键字。

本题需要考虑如何处理枚举合并顺序, 保证不漏的情况下尽量减少重复。

5.2.2 Code

```
1 class Solution {
2 public:
3     struct point{
4         int x;
5         int y;
6         point(int x,int y):x(x),y(y) {}
7         point(): x(-1),y(-1){}
8         bool operator==(const point& p)const{
9             return this->x==p.x && this->y==p.y;
10        }
11        bool operator<((const point&p )const{
12            if(this->x == p.x) return this->y<p.y;
13            else return this->x<p.x;
14        }
15        point(const point& p): x(p.x),y(p.y){}
16        void swap(point& p){
17            int temp_x = p.x;
18            int temp_y = p.y;
19            p.x = this->x;
20            p.y = this->y;
21            this->x = temp_x;
22            this->y = temp_y;
23        }
24        friend ostream& operator<<((ostream& out,const point& p)
25            {
```

```

25         return out<<p.x<<" "<<p.y<<" ";
26     }
27 };
28
29 struct DSU_size{
30     map<point,point>e;
31     DSU_size(vector<vector<char>>& grid){
32         for(int i = 0;i<grid.size();i++){
33             for(int j = 0;j<grid[0].size();j++){
34                 if(grid[i][j]=='1'){
35                     point p{i,j};
36                     point* l = new point();
37                     e[p] = *l;
38                 }
39             }
40         }
41     }
42     point find(point& p){return e[p].x<0?p:e[p] = find(e[p
43         ]);}
44     bool check_same(point& p1,point& p2){return find(p1)==
45         find(p2);}
46     bool size(point& p){return -find(p).x;}
47     int unite(int x1,int y1,int x2,int y2){
48         point p1{x1,y1};
49         point p2{x2,y2};
50         p1 = find(p1), p2 = find(p2);
51         if(p1==p2) return 0;
52         if(e[p1].x<e[p2].x) p1.swap(p2);
53         e[p2].x += e[p1].x;
54         e[p1] = p2;
55         return -1;
56     }
57 };
58
59 int numIslands(vector<vector<char>>& grid) {
60     DSU_size dsu(grid);
61     int n = grid.size();
62     int m = grid[0].size();

```

```
62     int land_cnt = 0;
63     for(int i = 0; i < n; i++){
64         for(int j = 0; j < m; j++){
65             land_cnt += grid[i][j] == '1' ? 1 : 0;
66         }
67     }
68     if(grid.size() == 0) return 0;
69     int merge_cnt = 0;
70     for(int i = 0; i < n; i++){
71         for(int j = 0; j < m; j++){
72             if(grid[i][j] != '1') continue;
73             if(i > 0 && grid[i-1][j] == '1') merge_cnt += dsu.
                unite(i, j, i-1, j);
74             if(j > 0 && grid[i][j-1] == '1') merge_cnt += dsu.
                unite(i, j, i, j-1);
75         }
76     }
77     // for(auto iter = dsu.e.begin(); iter != dsu.e.end(); iter
78     // ++){
79     //     cout << iter->first << iter->second << endl;
80     // }
81     return land_cnt + merge_cnt;
82 };
```

6 参考资料

- [1]. [USACO-Guide](#)
- [2]. [geeksforgeeks](#)
- [3]. [wikipedia](#)
- [4]. 算法导论 (原书第三版)

数学-恒等式

ZeitHaum

2023 年 4 月 12 日

目录

1
$$\sum_{i=1}^n \frac{1}{i} = \sum_{i=1}^n (-1)^{i-1} \frac{1}{i} \binom{n}{i}$$
1

$$1 \quad \sum_{I=1}^N \frac{1}{I} = \sum_{I=1}^N (-1)^{I-1} \frac{1}{I} \binom{N}{I}$$

$$1 \quad \sum_{i=1}^n \frac{1}{i} = \sum_{i=1}^n (-1)^{i-1} \frac{1}{i} \binom{n}{i}$$

差分法证明，记右式为 $f(n)$ ，左式为 $g(n)$ 。

$$\begin{aligned} f(n+1) - f(n) &= \sum_{i=1}^{n+1} (-1)^{i-1} \frac{1}{i} \binom{n+1}{i} - \sum_{i=1}^n (-1)^{i-1} \frac{1}{i} \binom{n}{i} \\ &= \sum_{i=1}^{n+1} (-1)^{i-1} \frac{1}{i} \left(\binom{n}{i} + \binom{n}{i-1} \right) - \sum_{i=1}^n (-1)^{i-1} \frac{1}{i} \binom{n}{i} \\ &= \sum_{i=1}^{n+1} (-1)^{i-1} \frac{1}{i} \binom{n}{i-1} + (-1)^{n+1} \frac{1}{n+1} \binom{n}{n+1} \\ &= \sum_{i=0}^n (-1)^i \frac{1}{i+1} \binom{n}{i} + 0 \\ &= \sum_{i=0}^{n+1} (-1)^i \frac{1}{i+1} \binom{n}{i} \end{aligned}$$

通过差分将上式化为比较简单的式子，接下来处理每一项中的 $\frac{1}{i+1}$ 部分，注意到

$$\begin{aligned} \frac{1}{i+1} \binom{n}{i} &= \frac{1}{i+1} \frac{n!}{(n-i)!i!} \\ &= \frac{n!}{(n-i)!(i+1)!} \\ &= \frac{(n+1)!}{(n-i)!(i+1)!} \frac{1}{n+1} \\ &= \binom{n+1}{i+1} \frac{1}{n+1} \end{aligned}$$

$$1 - \sum_{I=1}^N \frac{1}{I} = \sum_{I=1}^N (-1)^{I-1} \frac{1}{I} \binom{N}{I}$$

代回，得

$$\begin{aligned}
f(n+1) - f(n) &= \sum_{i=0}^{n+1} (-1)^i \frac{1}{i+1} \binom{n}{i} \\
&= \sum_0^{n+1} (-1)^i \frac{1}{n+1} \binom{n+1}{i+1} \\
&= \frac{1}{n+1} \left(\sum_1^{n+2} (-1)^{i-1} \binom{n+1}{i} - 1 + 1 \right) \\
&= \frac{1}{n+1} \left(\sum_1^{n+1} (-1)^{i-1} \binom{n+1}{i} + (-1)^{-1} \binom{n+1}{0} + 1 \right) \\
&= \frac{1}{n+1} \left(\sum_0^{n+1} (-1)^{i-1} \binom{n+1}{i} + 1 \right) \\
&= \frac{1}{n+1} \left(- \sum_0^{n+1} \binom{n+1}{i} (-1)^i 1^{n+1-i} + 1 \right) \\
&= \frac{1}{n+1} (-(-1+1)^{n+1} + 1) \\
&= \frac{1}{n+1}.
\end{aligned}$$

因此有 $f(n+1) - f(n) = g(n+1) - g(n)$.

考虑 $n=1$ 时, $f(1) = g(1) = 1$.

所以 $f(n) = g(n)$ 。

QED.

数学-数论

ZeitHaum

2023 年 4 月 26 日

目录

1 互质勾股数的生成方法	1
1.1 证明	1
1.2 启示	2
2 奇数平方和	2

1 互质勾股数的生成方法

问题描述: 构造以下集合: $S = \{(x^2 - y^2, 2xy, x^2 + y^2) | x > y \text{ 且 } x, y \in \mathbb{N}^+\}$, $S' = \{(a, b, c) | a^2 + b^2 = c^2 \text{ 且 } a, b, c \in \mathbb{N}^+\}$, 求证以下两个命题:

子命题 1. 在不考虑元组内元素顺序的情况下, 对于 S 中的任意元素 S_i , 都有 $S_i \in S'$.

子命题 2. 在不考虑元组内元素顺序的情况下, 对于 S' 中的任意元素 S'_i , 都存在 $k \in \mathbb{N}^+$ 使得 $kS'_i \in S$ 注: 对于 3 元组 (a, b, c) , 定义 $k(a, b, c) = (ka, kb, kc)$ 。

1.1 证明

对于命题 1, 有

$$\begin{aligned}(x^2 - y^2)^2 + (2xy)^2 &= (x^4 + y^4 - 2x^2y^2) + 4x^2y^2 \\ &= x^4 + y^4 + 2x^2y^2 \\ &= (x^2 + y^2)^2\end{aligned}$$

且显然 $x^2 - y^2, 2xy, x^2 + y^2 \in \mathbb{N}^+$, 于是子命题 1 得证。

对于命题 2, 令 $x = b, y = c - a, k = 2y = 2(c - a)$,

根据三角形性质, 显然有 $b > c - a, k \in \mathbb{N}^+$,

所以 $(x^2 - y^2, 2xy, x^2 + y^2) \in S$.

又

$$\begin{aligned}x^2 - y^2 &= b^2 - (c - a)^2 \\ &= (c^2 - a^2) - c^2 - a^2 + 2ac \\ &= 2ac - 2a^2 \\ &= ka,\end{aligned}$$

$$2xy = kb,$$

$$\begin{aligned}
x^2 + y^2 &= b^2 + (c - a)^2 \\
&= c^2 - a^2 + c^2 + a^2 - 2ac \\
&= 2c^2 - 2ac \\
&= kc.
\end{aligned}$$

所以 $(x^2 - y^2, 2xy, x^2 + y^2) = (ka, kb, kc) \in S'$, 证毕。

1.2 启示

以上两个定理说明通过 S 的构造方法可以显示的求出所有互质的勾股数, 只需枚举 x, y 求得 $(x^2 - y^2, 2xy, x^2 + y^2)$ 再让每个数除以三个数的最大公因数即可。

2 奇数平方和

求证: 两个奇数的平方和不可能为完全平方数。

证明:

可以通过反证法证明:

假设 a, b 都为奇数, 所以 c^2 为偶数。

不妨设 $a = 2p + 1, b = 2q + 1, (p, q \in \mathbb{N})$, 所以

$$a^2 = (2p + 1)^2 = 4p^2 + 4p + 1$$

所以 $a^2 \bmod 4 = 1, b^2 \bmod 4 = 1$.

所以 $a^2 + b^2 \bmod 4 = 2$.

又 $a^2 + b^2 = c^2$ 是完全平方数, 所以 $a^2 + b^2 \bmod 4 = 0$.

$a^2 + b^2 \bmod 4 = 2$ 和 $a^2 + b^2 \bmod 4 = 0$ 矛盾, 因此得证。