

Contents

1	Graph(No Weight)	1
1.1	SparseGraph	1
1.2	Path(DFS)	1
1.3	Component	2
1.4	ShortestPath(BFS)	2
2	Graph(Weight)	3
2.1	SparseGraph	3
3	Math	4
3.1	FindPrime	4
4	Heap	4
4.1	IndexMaxHeap	4

1 Graph(No Weight)

1.1 SparseGraph

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // 稀疏图 - 邻接表
5 class SparseGraph {
6     private:
7         int n, m;           // 节点数和边数
8         bool directed;       // 是否为有向图
9         vector<vector<int>>> g; // 图的具体数据
10    public:
11        // 构造函数
12        SparseGraph(int n, bool directed) {
13            assert(n >= 0);
14            this->n = n;
15            this->m = 0; // 初始化没有任何边
16            this->directed = directed;
17            // g初始化为n个空的vector,
18            // 表示每一个g[i]都为空, 即没有任和边
19            g = vector<vector<int>>>(n, vector<int>());
20        }
21        ~SparseGraph() {}
22        int V() { return n; } // 返回节点个数
23        int E() { return m; } // 返回边的个数
24        // 向图中添加一个边
25        void addEdge(int v, int w) {
26            g[v].push_back(w);
27            if (v != w && !directed)
28                g[w].push_back(v);
29            m++;
30        }
31        // 验证图中是否有从v到w的边
32        bool hasEdge(int v, int w) {
33            for (int i = 0; i < g[v].size(); i++)
34                if (g[v][i] == w)
35                    return true;
36            return false;
37        }
38        // 显示图的信息
39        void show() {
40            for (int i = 0; i < n; i++) {
41                cout << "vertex " << i << ": ";
42                for (int j = 0; j < g[i].size(); j++)
43                    cout << g[i][j] << " ";
44                cout << endl;
45            }
46        }
47        // 邻边迭代器, 传入一个图和一个顶点,
48        // 迭代在这个图中和这个顶点向连的所有顶点
49        class adjIterator {
50            private:
51                SparseGraph &G; // 图G的引用
52                int v;
53                int index;

```

```

53
54    public:
55        // 构造函数
56        adjIterator(SparseGraph &graph, int v) :
57            G(graph) {
58                this->v = v;
59                this->index = 0;
60            }
61        ~adjIterator() {}
62        // 返回图G中与顶点v相连接的第一个顶点
63        int begin() {
64            index = 0;
65            if (G.g[v].size())
66                return G.g[v][index];
67            // 若没有顶点和v相连接, 则返回 -1
68            return -1;
69        }
70        // 返回图G中与顶点v相连接的下一个顶点
71        int next() {
72            index++;
73            if (index < G.g[v].size())
74                return G.g[v][index];
75            // 若没有顶点和v相连接, 则返回 -1
76            return -1;
77        }
78        // 查看是否已经迭代完了图G中与顶点v相连接的所有顶点
79        bool end() {
80            return index >= G.g[v].size();
81        }
82    };

```

1.2 Path(DFS)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // visied from 陣列大小都是圖的頂點數量G.V()
5
6 // 路径查询
7 template <typename Graph>
8 class Path {
9     private:
10         Graph &G; // 图的引用
11         int s; // 起始点
12         bool *visited; // 记录dfs的过程中节点是否被访问
13         int *from; // 记录路径,
14         // from[i]表示查找的路径上i的上一个节点
15         // 图的深度优先遍历
16         void dfs(int v) {
17             visited[v] = true;
18             typename Graph::adjIterator adj(G, v);
19             for (int i = adj.begin(); !adj.end(); i = adj.next()) {
20                 if (!visited[i]) {
21                     from[i] = v;
22                     dfs(i);
23                 }
24             }
25         }
26    public:
27        // 构造函数, 寻路算法,
28        // 寻找图graph从s点到其他点的路径
29        Path(Graph &graph, int s) : G(graph) {
30            // 算法初始化
31            visited = new bool[G.V()];
32            from = new int[G.V()];
33            for (int i = 0; i < G.V(); i++) {
34                visited[i] = false;
35                from[i] = -1;
36            }
37            this->s = s;

```

```

37 // 寻路算法
38 dfs(s);
39 }
40
41 // 析构函数
42 ~Path() {
43     delete[] visited;
44     delete[] from;
45 }
46 // 查询从s点到w点是否有路径
47 bool hasPath(int w) {
48     assert(w >= 0 && w < G.V());
49     return visited[w];
50 }
51 // 查询从s点到w点的路径, 存放在vec中
52 void path(int w, vector<int> &vec) {
53     stack<int> s;
54     // 通过from数组逆向查找到从s到w的路径,
55     // 存放到栈中
56     int p = w;
57     while (p != -1) {
58         s.push(p);
59         p = from[p];
60     }
61     // 从栈中依次取出元素, 获得顺序的从s到w的路径
62     vec.clear();
63     while (!s.empty()) {
64         vec.push_back(s.top());
65         s.pop();
66     }
67
68 // 打印出从s点到w点的路径
69 void showPath(int w) {
70     vector<int> vec;
71     path(w, vec);
72     for (int i = 0; i < vec.size(); i++) {
73         cout << vec[i];
74         if (i == vec.size() - 1)
75             cout << endl;
76         else
77             cout << " -> ";
78     }
79 }
80 };

```

1.3 Component

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 //visited id 的大小都是Graph的頂點數量G.V()
5
6 //找連通分量
7 template <typename Graph>
8 class Component {
9     private:
10         Graph &G;
11         bool *visited;
12         int ccount = 0;
13         int *id;
14         void dfs(int v) {
15             visited[v] = true;
16             id[v] = ccount;
17             typename Graph::adjIterator adj(G, v);
18             for (int i = adj.begin(); !adj.end(); i = adj.next())
19                 if (!visited[i])
20                     dfs(i);
21         }
22     public:
23         Component(Graph &graph) : G(graph) {
24             visited = new bool[G.V()];
25             id = new int[G.V()];
26

```

```

27         for (int i = 0; i < G.V(); i++) {
28             visited[i] = false;
29             id[i] = -1;
30         }
31         ccount = 0;
32
33         for (int i = 0; i < G.V(); i++)
34             if (!visited[i]) {
35                 dfs(i);
36                 ccount += 1;
37             }
38     }
39     ~Component() {
40         delete[] visited;
41         delete[] id;
42     }
43     int count() {
44         return ccount;
45     }
46     bool isConnected(int v, int w) {
47         assert(v >= 0 && v < G.V());
48         assert(w >= 0 && w < G.V());
49         assert(id[v] != -1 && id[w] != -1);
50         return id[v] == id[w];
51     }
52 };

```

1.4 ShortestPath(BFS)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 //visited from ord 陣列大小都是圖的頂點數量G.V()
5
6 //寻找无权图的最短路径
7 template <typename Graph>
8 class ShortestPath {
9     private:
10         Graph &G; // 图的引用
11         int s; // 起始点
12         bool *visited; // 记录dfs的过程中节点是否被访问
13         int *from; // 记录路径,
14         // from[i]表示查找的路径上i的上一个节点
15         int *ord; //
16         // 记录路径中节点的次序。ord[i]表示i节点在路径中的次序。
17     public:
18         // 构造函数,
19         // 寻找无权图graph从s点到其他点的最短路径
20         ShortestPath(Graph &graph, int s) : G(graph) {
21             visited = new bool[graph.V()];
22             from = new int[graph.V()];
23             ord = new int[graph.V()];
24             for (int i = 0; i < graph.V(); i++) {
25                 visited[i] = false;
26                 from[i] = -1;
27                 ord[i] = -1;
28             }
29             this->s = s;
30             // 无向图最短路径算法,
31             // 从s开始广度优先遍历整张图
32             queue<int> q;
33             q.push(s);
34             visited[s] = true;
35             ord[s] = 0;
36             while (!q.empty()) {
37                 int v = q.front();
38                 q.pop();
39
40                 typename Graph::adjIterator adj(G, v);
41                 for (int i = adj.begin(); !adj.end(); i = adj.next())
42                     if (!visited[i]) {
43                         q.push(i);
44

```

```

41         visited[i] = true;
42         from[i] = v;
43         ord[i] = ord[v] + 1;
44     }
45 }
46
47 // 析构函数
48 ~ShortestPath() {
49     delete[] visited;
50     delete[] from;
51     delete[] ord;
52 }
53
54 // 查询从s点到w点是否有路径
55 bool hasPath(int w) {
56     return visited[w];
57 }
58
59 // 查询从s点到w点的路径，存放在vec中
60 void path(int w, vector<int> &vec) {
61     stack<int> s;
62     // 通过from数组逆向查找到从s到w的路径，
63     // 存放到栈中
64     int p = w;
65     while (p != -1) {
66         s.push(p);
67         p = from[p];
68     }
69     // 从栈中依次取出元素，获得顺序的从s到w的路径
70     vec.clear();
71     while (!s.empty()) {
72         vec.push_back(s.top());
73         s.pop();
74     }
75 }
76
77 // 打印出从s点到w点的路径
78 void showPath(int w) {
79     vector<int> vec;
80     path(w, vec);
81     for (int i = 0; i < vec.size(); i++) {
82         cout << vec[i];
83         if (i == vec.size() - 1)
84             cout << endl;
85         else
86             cout << " -> ";
87     }
88 }
89
90 // 查看从s点到w点的最短路径长度
91 int length(int w) {
92     assert(w >= 0 && w < G.V());
93     return ord[w];
94 }
95 };

```

2 Graph(Weight)

2.1 SparseGraph

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // 边
5 template <typename Weight>
6 class Edge {
7     // 输出边的信息
8     friend ostream& operator<<(ostream& os, const
9         Edge& e) {
10         os << e.a << " -> " << e.b << " : " << e.weight;
11         return os;
12     }
13 private:

```

```

14     int a, b;           // 边的两个端点
15     Weight weight;      // 边的权值
16 public:
17     // 构造函数
18     Edge(int a, int b, Weight weight) {
19         this->a = a;
20         this->b = b;
21         this->weight = weight;
22     }
23     // 空的构造函数，所有的成员变量都取默认值
24     Edge() {}
25
26     ~Edge() {}
27
28     int v() { return a; } // 返回第一个顶点
29     int w() { return b; } // 返回第二个顶点
30     Weight wt() { return weight; } // 返回权值
31     // 给定一个顶点，返回另一个顶点
32     int other(int x) {
33         assert(x == a || x == b);
34         return x == a ? b : a;
35     }
36     // 边的大小比较，是对边的权值的大小比较
37     bool operator<(Edge<Weight>& e) {
38         return weight < e.wt();
39     }
40     bool operator<=(Edge<Weight>& e) {
41         return weight <= e.wt();
42     }
43     bool operator>(Edge<Weight>& e) {
44         return weight > e.wt();
45     }
46     bool operator>=(Edge<Weight>& e) {
47         return weight >= e.wt();
48     }
49     bool operator==(Edge<Weight>& e) {
50         return weight == e.wt();
51     }
52 };

```

```

1 #include <bits/stdc++.h>
2
3 #include "Edge.h"
4 using namespace std;
5
6 // 稀疏图 - 邻接表
7 template <typename Weight>
8 class SparseGraph {
9     private:
10         int n, m; // 节点数和边数
11         bool directed; // 是否为有向图
12         vector<vector<Edge<Weight> *> > g; // 图的具体数据
13     public:
14         // 构造函数
15         SparseGraph(int n, bool directed) {
16             assert(n >= 0);
17             this->n = n;
18             this->m = 0; // 初始化没有任何边
19             this->directed = directed;
20             // g初始化为n个空的vector，
21             // 表示每一个g[i]都为空，即没有任和边
22             g = vector<vector<Edge<Weight> *> >(n,
23                 vector<Edge<Weight> *>());
24         }
25         // 析构函数
26         ~SparseGraph() {
27             for (int i = 0; i < n; i++)
28                 for (int j = 0; j < g[i].size(); j++)
29                     delete g[i][j];
30         }
31         int V() { return n; } // 返回节点个数
32         int E() { return m; } // 返回边的个数

```

```

31 // 向图中添加一个边, 权值为 weight
32 void addEdge(int v, int w, Weight weight) {
33     // 注意, 由于在邻接表的情况,
34     // 查找是否有重边需要遍历整个链表
35     // 我们的程序允许重边的出现
36     g[v].push_back(new Edge<Weight>(v, w,
37     weight));
38     if (v != w && !directed)
39         g[w].push_back(new Edge<Weight>(w, v,
40         weight));
41     m++;
42 }
43 // 验证图中是否有从 v 到 w 的边
44 bool hasEdge(int v, int w) {
45     for (int i = 0; i < g[v].size(); i++)
46         if (g[v][i]->other(v) == w)
47             return true;
48     return false;
49 }
50 // 显示图的信息
51 void show() {
52     for (int i = 0; i < n; i++) {
53         cout << "vertex " << i << ": \t";
54         for (int j = 0; j < g[i].size(); j++)
55             cout << "( to: " << g[i][j]->w() <<
56             ", wt: " << g[i][j]->wt() << ") \t";
57         cout << endl;
58     }
59 }
60 // 邻边迭代器, 传入一个图和一个顶点,
61 // 迭代在这个图中和这个顶点向连的所有边
62 class adjIterator {
63 private:
64     SparseGraph &G; // 图 G 的引用
65     int v;
66     int index;
67 public:
68     // 构造函数
69     adjIterator(SparseGraph &graph, int v) :
70         G(graph) {
71         this->v = v;
72         this->index = 0;
73     }
74     ~adjIterator() {}
75     // 返回图 G 中与顶点 v 相连接的第一个边
76     Edge<Weight> *begin() {
77         index = 0;
78         if (G.g[v].size())
79             return G.g[v][index];
80         // 若没有顶点和 v 相连接, 则返回 NULL
81         return NULL;
82     }
83     // 返回图 G 中与顶点 v 相连接的下一个边
84     Edge<Weight> *next() {
85         index += 1;
86         if (index < G.g[v].size())
87             return G.g[v][index];
88         return NULL;
89     }
90     // 查看是否已经迭代完了图 G 中与顶点 v 相连接的所有边
91     bool end() {
92         return index >= G.g[v].size();
93     }
94 };

```

3 Math

3.1 FindPrime

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // 查找 [0, 2^15] 中的所有质数 共有 3515
5
6 const int MAXN = 32768; // 2^15=32768
7 bool primes[MAXN];
8 vector<int> p; // 3515
9
10 // 质数筛法 Sieve of Eratosthenes
11 inline void findPrimes() {
12     for (int i = 0; i < MAXN; i++) {
13         primes[i] = true;
14     }
15     primes[0] = false;
16     primes[1] = false;
17     for (int i = 4; i < MAXN; i += 2) {
18         // 将 2 的倍数全部删掉 (偶数不会是质数)
19         primes[i] = false;
20     }
21     // 开始逐个检查 ---> 小心 i*i 会有 overflow 问题 ---> 使用 long
22     long
23     for (long long i = 3; i < MAXN; i += 2) {
24         if (primes[i]) {
25             // 如果之前还未被删掉 才做筛法
26             for (long long j = i * i; j < MAXN; j +=
27                 i) {
28                 // 从 i*i 开始 (因为 i*2, i*3... 都被前面处理完了)
29                 primes[j] = false;
30             }
31         }
32     }
33     // 蒐集所有质数
34     for (int i = 0; i < MAXN; i++) {
35         if (primes[i]) {
36             p.emplace_back(i);
37         }
38     }
39 }

```

4 Heap

4.1 IndexMaxHeap

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // data indexes reverse
5 // 陣列邊界都是以 [1, capacity] 處理 ---> 陣列大小為 (capacity+1)
6
7 // 最大索引堆
8 template <typename Item>
9 class IndexMaxHeap {
10 private:
11     Item *data; // 最大索引堆中的数据
12     int *indexes; // 最大索引堆中的索引, indexes[x]
13     // = i 表示索引 i 在 x 的位置
14     int *reverse; // 最大索引堆中的反向索引,
15     // reverse[i] = x 表示索引 i 在 x 的位置
16     int count;
17     int capacity;
18     // 索引堆中,
19     // 数据之间的比较根据 data 的大小进行比较,
20     // 但实际操作的是索引
21     void shiftUp(int k) {
22         while (k > 1 && data[indexes[k / 2]] <
23             data[indexes[k]]) {
24             swap(indexes[k / 2], indexes[k]);
25             reverse[indexes[k / 2]] = k / 2;
26             reverse[indexes[k]] = k;
27             k /= 2;
28         }
29     }

```

```

22     }
23 }
24 // 索引堆中,
    数据之间的比较根据data的大小进行比较,
    但实际操作的是索引
25 void shiftDown(int k) {
26     while (2 * k <= count) {
27         int j = 2 * k;
28         if (j + 1 <= count && data[indexes[j] +
29             1] > data[indexes[j]])
30             j += 1;
31
32         if (data[indexes[k]] >= data[indexes[j]])
33             break;
34
35         swap(indexes[k], indexes[j]);
36         reverse[indexes[k]] = k;
37         reverse[indexes[j]] = j;
38         k = j;
39     }
40
41 public:
42 // 构造函数, 构造一个空的索引堆,
    可容纳capacity个元素
43 IndexMaxHeap(int capacity) {
44     data = new Item[capacity + 1];
45     indexes = new int[capacity + 1];
46     reverse = new int[capacity + 1];
47     for (int i = 0; i <= capacity; i++)
48         reverse[i] = 0;
49
50     count = 0;
51     this->capacity = capacity;
52 }
53 ~IndexMaxHeap() {
54     delete[] data;
55     delete[] indexes;
56     delete[] reverse;
57 }
58 // 返回索引堆中的元素个数
59 int size() {
60     return count;
61 }
62 // 返回一个布尔值, 表示索引堆中是否为空
63 bool isEmpty() {
64     return count == 0;
65 }
66 // 向最大索引堆中插入一个新的元素,
    新元素的索引为i, 元素为item
67 // 传入的i对用户而言, 是从0索引的
68 void insert(int i, Item item) {
69     i += 1;
70     data[i] = item;
71     indexes[count + 1] = i;
72     reverse[i] = count + 1;
73     count++;
74     shiftUp(count);
75 }
76
77 // 从最大索引堆中取出堆顶元素,
    即索引堆中所存储的最大数据
78 Item extractMax() {
79     Item ret = data[indexes[1]];
80     swap(indexes[1], indexes[count]);
81     reverse[indexes[count]] = 0;
82     count--;
83
84     if (count) {
85         reverse[indexes[1]] = 1;
86         shiftDown(1);
87     }
88
89     return ret;
90 }
91
92 // 从最大索引堆中取出堆顶元素的索引
93 int extractMaxIndex() {
94     int ret = indexes[1] - 1;
95     swap(indexes[1], indexes[count]);
96     reverse[indexes[count]] = 0;
97     count--;
98     if (count) {
99         reverse[indexes[1]] = 1;
100        shiftDown(1);
101    }
102    return ret;
103 }
104 // 获取最大索引堆中的堆顶元素
105 Item getMax() {
106     return data[indexes[1]];
107 }
108 // 获取最大索引堆中的堆顶元素的索引
109 int getMaxIndex() {
110     return indexes[1] - 1;
111 }
112 // 看索引i所在的位置是否存在元素
113 bool contain(int i) {
114     return reverse[i + 1] != 0;
115 }
116 // 获取最大索引堆中索引为i的元素
117 Item getItem(int i) {
118     return data[i + 1];
119 }
120
121 // 将最大索引堆中索引为i的元素修改为newItem
122 void change(int i, Item newItem) {
123     i += 1;
124     data[i] = newItem;
125     shiftUp(reverse[i]);
126     shiftDown(reverse[i]);
127 }
128 };

```