

## 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
3	Math	3
3.1	FindPrime	3
4	Heap	3
4.1	IndexMaxHeap	3

## 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 << ": \t";
42                for (int j = 0; j < g[i].size(); j++)
43                    cout << g[i][j] << " \t";
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;

```

```

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;
38            // 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
23 public:
24     Component(Graph &graph) : G(graph) {
25         visited = new bool[G.V()];
26         id = new int[G.V()];
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; // 记录路径中节点的次序。ord[i]表示i节点在路径中的次序。
16
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[G.V()];
22         from = new int[G.V()];
23         ord = new int[G.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                     visited[i] = true;

```

```

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

```

## 2 Graph(Weight)

## 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
        long
22     for (long long i = 3; i < MAXN; i += 2) {
23         if (primes[i]) {
24             //如果之前还未被删掉 才做筛法
25             for (long long j = i * i; j < MAXN; j += i) {
26                 //从i*i开始(因为i*2,i*3...都被前面处理完了)
27                 primes[j] = false;
28             }
29         }
30     }
31     //蒐集所有质数
32     for (int i = 0; i < MAXN; i++) {
33         if (primes[i]) {
34             p.emplace_back(i);
35         }
36     }
37 }
38 }

```

## 4 Heap

### 4.1 IndexMaxHeap

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 //data indexes reverse
        陣列邊界都是以[1, capacity]處理--->陣列大小為(capacity+1)
5
6 // 最大索引堆
7 template <typename Item>
8 class IndexMaxHeap {
9     private:
10         Item *data; // 最大索引堆中的数据
11         int *indexes; // 最大索引堆中的索引, indexes[x]
            = i 表示索引i在x的位置
12         int *reverse; // 最大索引堆中的反向索引,
            reverse[i] = x 表示索引i在x的位置
13         int count;
14         int capacity;
15         // 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
16         void shiftUp(int k) {
17             while (k > 1 && data[indexes[k / 2]] <
                data[indexes[k]]) {
18                 swap(indexes[k / 2], indexes[k]);
19                 reverse[indexes[k / 2]] = k / 2;
20                 reverse[indexes[k]] = k;
21                 k /= 2;
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] +
                    1] > data[indexes[j]])
29                     j += 1;
30             }

```

```

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

```