

Contents

1	Math	1
1.1	FindPrime	1
2	Graph(No Weight)	1
2.1	SparseGraph	1
2.2	Path(DFS)	2
2.3	Component	2
2.4	ShortestPath(BFS)	3
3	Graph(Weight)	3
3.1	SparseGraph	3
3.2	MinimumSpanTree	4
3.3	ShortestPath	5
4	Heap	7
4.1	IndexMaxHeap	7

1 Math

1.1 FindPrime

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 //查找[0,2^15]中的所有質數 共有3515
5
6 const int MAXN = 32768; //2^15=32768
7 bool primes[MAXN];
8 vector<int> p; //3515
9
10 //質數篩法 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         //將2的倍數全部刪掉(偶數不會是質數)
19         primes[i] = false;
20     }
21     //開始逐個檢查--->小心i*i會有overflow問題--->使用long
22     long
23     for (long long i = 3; i < MAXN; i += 2) {
24         if (primes[i]) {
25             //如果之前還未被刪掉 才做篩法
26             for (long long j = i * i; j < MAXN; j += i) {
27                 //從i*i開始(因為i*2,i*3...都被前面處理完)
28                 primes[j] = false;
29             }
30         }
31     }
32     //蒐集所有質數
33     for (int i = 0; i < MAXN; i++) {
34         if (primes[i]) {
35             p.emplace_back(i);
36         }
37     }
38 }

```

2 Graph(No Weight)

2.1 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            }
79        }

```

```

77 // 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
78 bool end() {
79     return index >= G.g[v].size();
80 }
81 };
82 };

```

2.2 Path(DFS)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 //visited 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
39             dfs(s);
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             }

```

```

60 // 从栈中依次取出元素, 获得顺序的从s到w的路径
61 vec.clear();
62 while (!s.empty()) {
63     vec.push_back(s.top());
64     s.pop();
65 }
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 };

```

2.3 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         }
40         ~Component() {
41             delete[] visited;
42             delete[] id;
43         }
44         int count() {
45             return ccount;
46         }
47         bool isConnected(int v, int w) {
48             assert(v >= 0 && v < G.V());
49             assert(w >= 0 && w < G.V());
50             assert(id[v] != -1 && id[w] != -1);
51             return id[v] == id[w];

```

```
51 }
52 };
```

2.4 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     public:
17         // 构造函数,
18         // 寻找无权图graph从s点到其他点的最短路径
19         ShortestPath(Graph &graph, int s) : G(graph) {
20             visited = new bool[graph.V()];
21             from = new int[graph.V()];
22             ord = new int[graph.V()];
23             for (int i = 0; i < graph.V(); i++) {
24                 visited[i] = false;
25                 from[i] = -1;
26                 ord[i] = -1;
27             }
28             this->s = s;
29             // 无向图最短路径算法,
30             // 从s开始广度优先遍历整张图
31             queue<int> q;
32             q.push(s);
33             visited[s] = true;
34             ord[s] = 0;
35             while (!q.empty()) {
36                 int v = q.front();
37                 q.pop();
38
39                 typename Graph::adjIterator adj(G, v);
40                 for (int i = adj.begin(); !adj.end(); i = adj.next())
41                     if (!visited[i]) {
42                         q.push(i);
43                         visited[i] = true;
44                         from[i] = v;
45                         ord[i] = ord[v] + 1;
46                     }
47             }
48
49             // 析构函数
50             ~ShortestPath() {
51                 delete[] visited;
52                 delete[] from;
53                 delete[] ord;
54             }
55
56             // 查询从s点到w点是否有路径
57             bool hasPath(int w) {
58                 return visited[w];
59             }
60
61             // 查询从s点到w点的路径, 存放在vec中
62             void path(int w, vector<int> &vec) {
63                 stack<int> s;
64                 // 通过from数组逆向查找到从s到w的路径,
65                 // 存放到栈中
66                 int p = w;
67                 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 };
```

3 Graph(Weight)

3.1 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
14     private:
15         int a, b;          // 边的两个端点
16         Weight weight;     // 边的权值
17     public:
18         // 构造函数
19         Edge(int a, int b, Weight weight) {
20             this->a = a;
21             this->b = b;
22             this->weight = weight;
23         }
24         // 空的构造函数, 所有的成员变量都取默认值
25         Edge() {}
26
27         ~Edge() {}
28
29         int v() { return a; } // 返回第一个顶点
30         int w() { return b; } // 返回第二个顶点
31         Weight wt() { return weight; } // 返回权值
32         // 给定一个顶点, 返回另一个顶点
33         int other(int x) {
34             assert(x == a || x == b);
35             return x == a ? b : a;
36         }
37         // 边的大小比较, 是对边的权值的大小比较
38         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
33         // 向图中添加一个边, 权值为weight
34         void addEdge(int v, int w, Weight weight) {
35             // 注意, 由于在邻接表的情况,
36             // 查找是否有重边需要遍历整个链表
37             // 我们的程序允许重边的出现
38             g[v].push_back(new Edge<Weight>(v, w,
39                 weight));
40             if (v != w && !directed)
41                 g[w].push_back(new Edge<Weight>(w, v,
42                     weight));
43             m++;
44         }
45         // 验证图中是否有从v到w的边
46         bool hasEdge(int v, int w) {
47             for (int i = 0; i < g[v].size(); i++)
48                 if (g[v][i]->other(v) == w)
49                     return true;
50             return false;
51         }
52         // 显示图的信息
53         void show() {
54             for (int i = 0; i < n; i++) {
55                 cout << "vertex " << i << ":\t";
56                 for (int j = 0; j < g[i].size(); j++)

```

```

53         cout << "( to:" << g[i][j]->w() <<
54             ",wt:" << g[i][j]->wt() << ")\t";
55         cout << endl;
56     }
57 }
58 // 邻边迭代器, 传入一个图和一个顶点,
59 // 迭代在这个图中和这个顶点向连的所有边
60 class adjIterator {
61     private:
62         SparseGraph &G; // 图G的引用
63         int v;
64         int index;
65     public:
66         // 构造函数
67         adjIterator(SparseGraph &graph, int v) :
68             G(graph) {
69             this->v = v;
70             this->index = 0;
71         }
72         ~adjIterator() {}
73         // 返回图G中与顶点v相连接的第一个边
74         Edge<Weight> *begin() {
75             index = 0;
76             if (G.g[v].size())
77                 return G.g[v][index];
78             // 若没有顶点和v相连接, 则返回NULL
79             return NULL;
80         }
81         // 返回图G中与顶点v相连接的下一个边
82         Edge<Weight> *next() {
83             index += 1;
84             if (index < G.g[v].size())
85                 return G.g[v][index];
86             return NULL;
87         }
88         // 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
89         bool end() {
90             return index >= G.g[v].size();
91         }
92 };

```

3.2 MinimumSpanTree

```

1 #include <bits/stdc++.h>
2
3 #include "Edge.h"
4 using namespace std;
5
6 // 陣列大小為傳入的數量n(通常是圖的頂點數量)
7 // Quick Union + rank + path compression
8 class UnionFind {
9     private:
10         int* rank;
11         int* parent; // parent[i]表示第i个元素所指向的父节点
12         int count; // 数据个数
13     public:
14         // 构造函数
15         UnionFind(int count) {
16             parent = new int[count];
17             rank = new int[count];
18             this->count = count;
19             for (int i = 0; i < count; i++) {
20                 parent[i] = i;
21                 rank[i] = 1;
22             }
23         }
24         // 析构函数
25         ~UnionFind() {
26             delete[] parent;
27             delete[] rank;

```

```

28 }
29 // 查找过程, 查找元素p所对应的集合编号
30 // O(h)复杂度, h为树的高度
31 int find(int p) {
32     // path compression 1
33     while (p != parent[p]) {
34         parent[p] = parent[parent[p]];
35         p = parent[p];
36     }
37     return p;
38 }
39 // 查看元素p和元素q是否所属一个集合
40 // O(h)复杂度, h为树的高度
41 bool isConnected(int p, int q) {
42     return find(p) == find(q);
43 }
44 // 合并元素p和元素q所属的集合
45 // O(h)复杂度, h为树的高度
46 void unionElements(int p, int q) {
47     int pRoot = find(p);
48     int qRoot = find(q);
49     if (pRoot == qRoot)
50         return;
51     //
52     // 根据两个元素所在树的元素个数不同判断合并方向
53     // 将元素个数少的集合合并到元素个数多的集合上
54     if (rank[pRoot] < rank[qRoot]) {
55         parent[pRoot] = qRoot;
56     } else if (rank[qRoot] < rank[pRoot]) {
57         parent[qRoot] = pRoot;
58     } else { // rank[pRoot] == rank[qRoot]
59         parent[pRoot] = qRoot;
60         rank[qRoot] += 1; // 此时, 我维护rank的值
61     }
62 };
63
64 // Kruskal算法
65 template <typename Graph, typename Weight>
66 class KruskalMST {
67 private:
68     vector<Edge<Weight>> mst; //
69     // 最小生成树所包含的所有边
70     Weight mstWeight; // 最小生成树的权值
71 public:
72     // 构造函数, 使用Kruskal算法计算graph的最小生成树
73     KruskalMST(Graph& graph) {
74         // 将图中的所有边存放到一个最小堆中
75         priority_queue<Edge<Weight>> pq(Edge<Weight>,
76             vector<Edge<Weight>>,
77             greater<Edge<Weight>>);
78         for (int i = 0; i < graph.V(); i++) {
79             typename Graph::adjIterator adj(graph, i);
80             for (Edge<Weight>* e = adj.begin();
81                 !adj.end(); e = adj.next())
82                 if (e->v() < e->w())
83                     pq.push(*e);
84         }
85         // 创建一个并查集,
86         // 来查看已经访问的节点的联通情况
87         UnionFind uf = UnionFind(graph.V());
88         while (!pq.isEmpty() && mst.size() <
89             graph.V() - 1) {
90             // 从最小堆中依次从小到大取出所有的边
91             Edge<Weight> e = pq.pop();
92             pq.pop();
93             // 如果该边的两个端点是联通的,
94             // 说明加入这条边将产生环, 扔掉这条边
95             if (uf.isConnected(e.v(), e.w()))
96                 continue;
97
98             // 否则, 将这条边添加进最小生成树,
99             // 同时标记边的两个端点联通

```

```

94         mst.push_back(e);
95         uf.unionElements(e.v(), e.w());
96     }
97
98     mstWeight = mst[0].wt();
99     for (int i = 1; i < mst.size(); i++)
100         mstWeight += mst[i].wt();
101 }
102 ~KruskalMST() {}
103 // 返回最小生成树的所有边
104 vector<Edge<Weight>> mstEdges() {
105     return mst;
106 };
107 // 返回最小生成树的权值
108 Weight result() {
109     return mstWeight;
110 };
111 };

```

3.3 ShortestPath

```

1 #include <bits/stdc++.h>
2
3 #include "Edge.h"
4
5 using namespace std;
6
7 //data indexes reverse 陣列大小都是capacity大小+1
8
9 // 最小索引堆
10 template <typename Item>
11 class IndexMinHeap {
12 private:
13     Item *data; // 最小索引堆中的数据
14     int *indexes; // 最小索引堆中的索引, indexes[x]
15     // = i 表示索引i在x的位置
16     int *reverse; // 最小索引堆中的反向索引,
17     // reverse[i] = x 表示索引i在x的位置
18     int count;
19     int capacity;
20     // 索引堆中,
21     // 数据之间的比较根据data的大小进行比较,
22     // 但实际操作的是索引
23     void shiftUp(int k) {
24         while (k > 1 && data[indexes[k / 2]] >
25             data[indexes[k]]) {
26             swap(indexes[k / 2], indexes[k]);
27             reverse[indexes[k / 2]] = k / 2;
28             reverse[indexes[k]] = k;
29             k /= 2;
30         }
31     }
32     // 索引堆中,
33     // 数据之间的比较根据data的大小进行比较,
34     // 但实际操作的是索引
35     void shiftDown(int k) {
36         while (2 * k <= count) {
37             int j = 2 * k;
38             if (j + 1 <= count && data[indexes[j]] >
39                 data[indexes[j + 1]])
40                 j += 1;
41             if (data[indexes[k]] <= data[indexes[j]])
42                 break;
43             swap(indexes[k], indexes[j]);
44             reverse[indexes[k]] = k;
45             reverse[indexes[j]] = j;
46             k = j;
47         }
48     }
49 public:
50     // 构造函数, 构造一个空的索引堆,
51     // 可容纳capacity个元素
52     IndexMinHeap(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     count = 0;
51     this->capacity = capacity;
52 }
53 ~IndexMinHeap() {
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 // 向最小索引堆中插入一个新的元素,
67 // 新元素的索引为i, 元素为item
68 // 传入的i对用户而言, 是从0索引的
69 void insert(int index, Item item) {
70     index += 1;
71     data[index] = item;
72     indexes[count + 1] = index;
73     reverse[index] = count + 1;
74     count++;
75     shiftUp(count);
76 }
77 // 从最小索引堆中取出堆顶元素,
78 // 即索引堆中所存储的最小数据
79 Item extractMin() {
80     Item ret = data[indexes[1]];
81     swap(indexes[1], indexes[count]);
82     reverse[indexes[count]] = 0;
83     reverse[indexes[1]] = 1;
84     count--;
85     shiftDown(1);
86     return ret;
87 }
88 // 从最小索引堆中取出堆顶元素的索引
89 int extractMinIndex() {
90     int ret = indexes[1] - 1;
91     swap(indexes[1], indexes[count]);
92     reverse[indexes[count]] = 0;
93     reverse[indexes[1]] = 1;
94     count--;
95     shiftDown(1);
96     return ret;
97 }
98
99 // 获取最小索引堆中的堆顶元素
100 Item getMin() {
101     return data[indexes[1]];
102 }
103
104 // 获取最小索引堆中的堆顶元素的索引
105 int getMinIndex() {
106     return indexes[1] - 1;
107 }
108
109 // 看索引i所在的位置是否存在元素
110 bool contain(int index) {
111     return reverse[index + 1] != 0;
112 }
113
114 // 获取最小索引堆中索引为i的元素
115 Item getItem(int index) {
116     return data[index + 1];
117 }
118
119 // 将最小索引堆中索引为i的元素修改为newItem
120 void change(int index, Item newItem) {
121     index += 1;
122     data[index] = newItem;
123     shiftUp(reverse[index]);
124     shiftDown(reverse[index]);
125 }
126 };
127
128 // Dijkstra算法求最短路径
129 template <typename Graph, typename Weight>
130 class Dijkstra {
131 private:
132     Graph &G; // 图的引用
133     int s; // 起始点
134     Weight *distTo; //
135     // distTo[i]存储从起始点s到i的最短路径长度
136     bool *marked; // 标记数组,
137     // 在算法运行过程中标记节点i是否被访问
138     vector<Edge<Weight> *> from; //
139     // from[i]记录最短路径中, 到达i点的边是哪一条
140 public:
141     // 构造函数, 使用Dijkstra算法求最短路径
142     Dijkstra(Graph &graph, int s) : G(graph) {
143         // 算法初始化
144         assert(s >= 0 && s < G.V());
145         this->s = s;
146         distTo = new Weight[G.V()];
147         marked = new bool[G.V()];
148         for (int i = 0; i < G.V(); i++) {
149             distTo[i] = Weight();
150             marked[i] = false;
151             from.push_back(NULL);
152         }
153         //
154         // 使用索引堆记录当前找到的到达每个顶点的最短距离
155         IndexMinHeap<Weight> ipq(G.V());
156         // 对于其实点s进行初始化
157         distTo[s] = Weight();
158         from[s] = new Edge<Weight>(s, s, Weight());
159         ipq.insert(s, distTo[s]);
160         marked[s] = true;
161         while (!ipq.isEmpty()) {
162             int v = ipq.extractMinIndex();
163             // distTo[v]就是s到v的最短距离
164             marked[v] = true;
165             // 对v的所有相邻节点进行更新
166             typename Graph::adjIterator adj(G, v);
167             for (Edge<Weight> *e = adj.begin();
168                  !adj.end(); e = adj.next()) {
169                 int w = e->other(v);
170                 // 如果从s点到w点的最短路径还没有找到
171                 if (!marked[w]) {
172                     // 如果w点以前没有访问过,
173                     // 或者访问过,
174                     // 但是通过当前的v点到w点距离更短,
175                     // 则进行更新
176                     if (from[w] == NULL || distTo[v]
177                         + e->wt() < distTo[w]) {
178                         distTo[w] = distTo[v] +
179                             e->wt();
180                         from[w] = e;
181                         if (ipq.contain(w))
182                             ipq.change(w, distTo[w]);
183                         else
184                             ipq.insert(w, distTo[w]);
185                     }
186                 }
187             }
188         }
189     }
190
191     // 析构函数
192     ~Dijkstra() {
193         delete[] distTo;
194     }
195 };

```



```

185     delete[] marked;
186     delete from[0];
187 }
188
189 // 返回从s点到w点的最短路径长度
190 Weight shortestPathTo(int w) {
191     return distTo[w];
192 }
193 // 判断从s点到w点是否联通
194 bool hasPathTo(int w) {
195     return marked[w];
196 }
197 // 寻找从s到w的最短路径,
198 // 将整个路径经过的边存放在vec中
199 void shortestPath(int w, vector<Edge<Weight>>
200 &vec) {
201     // 通过from数组逆向查找到从s到w的路径,
202     // 存放到栈中
203     stack<Edge<Weight>*> s;
204     Edge<Weight> *e = from[w];
205     while (e->v() != this->s) {
206         s.push(e);
207         e = from[e->v()];
208     }
209     s.push(e);
210     // 从栈中依次取出元素, 获得顺序的从s到w的路径
211     while (!s.empty()) {
212         e = s.top();
213         vec.push_back(*e);
214         s.pop();
215     }
216 // 打印出从s点到w点的路径
217 void showPath(int w) {
218     vector<Edge<Weight>> vec;
219     shortestPath(w, vec);
220     for (int i = 0; i < vec.size(); i++) {
221         cout << vec[i].v() << " -> ";
222         if (i == vec.size() - 1)
223             cout << vec[i].w() << endl;
224     }
225 };

```

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)
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 }
30 // 索引堆中,
31 // 数据之间的比较根据data的大小进行比较,
32 // 但实际操作的是索引
33 void shiftDown(int k) {
34     while (2 * k <= count) {
35         int j = 2 * k;
36         if (j + 1 <= count && data[indexes[j + 1]] > data[indexes[j]])
37             j += 1;
38         if (data[indexes[k]] >= data[indexes[j]])
39             break;
40         swap(indexes[k], indexes[j]);
41         reverse[indexes[k]] = k;
42         reverse[indexes[j]] = j;
43         k = j;
44     }
45 }
46 public:
47 // 构造函数, 构造一个空的索引堆,
48 // 可容纳capacity个元素
49 IndexMaxHeap(int capacity) {
50     data = new Item[capacity + 1];
51     indexes = new int[capacity + 1];
52     reverse = new int[capacity + 1];
53     for (int i = 0; i <= capacity; i++)
54         reverse[i] = 0;
55
56     count = 0;
57     this->capacity = capacity;
58 }
59 ~IndexMaxHeap() {
60     delete[] data;
61     delete[] indexes;
62     delete[] reverse;
63 }
64 // 返回索引堆中的元素个数
65 int size() {
66     return count;
67 }
68 // 返回一个布尔值, 表示索引堆中是否为空
69 bool isEmpty() {
70     return count == 0;
71 }
72 // 向最大索引堆中插入一个新的元素,
73 // 新元素的索引为i, 元素为item
74 // 传入的i对用户而言,是从0索引的
75 void insert(int i, Item item) {
76     i += 1;
77     data[i] = item;
78     indexes[count + 1] = i;
79     reverse[i] = count + 1;
80     count++;
81     shiftUp(count);
82 }
83
84 // 从最大索引堆中取出堆顶元素,
85 // 即索引堆中所存储的最大数据
86 Item extractMax() {
87     Item ret = data[indexes[1]];
88     swap(indexes[1], indexes[count]);
89     reverse[indexes[count]] = 0;
90     count--;
91
92     if (count) {
93         reverse[indexes[1]] = 1;
94         shiftDown(1);
95     }
96 }

```

```

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 };

```