

Contents

1	Math	1
1.1	FindPrime	1
2	Heap	1
2.1	IndexMaxHeap	1
3	Graph	2
3.1	SparseGraph	2
3.2	Path(DFS)	3
3.3	Component	3
3.4	ShortestPath(BFS)	3

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     for (long long i = 3; i < MAXN; i += 2) {
23         if (primes[i]) {
24             //如果之前還未被刪掉 才做篩法
25             for (long long j = i * i; j < MAXN; j += i) {
26                 //從i*i開始(因為i*2,i*3...都被前面處理完)
27                 primes[j] = false;
28             }
29         }
30     }
31     //蒐集所有質數
32     for (int i = 0; i < MAXN; i++) {
33         if (primes[i]) {
34             p.emplace_back(i);
35         }
36     }
37 }

```

2 Heap

2.1 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 // 最大索引堆
7 template <typename Item>

```

```

8 class IndexMaxHeap {
9     private:
10         Item *data; // 最大索引堆中的數據
11         int *indexes; // 最大索引堆中的索引, indexes[x]
12                        // = i 表示索引i在x的位置
13         int *reverse; // 最大索引堆中的反向索引,
14                        // reverse[i] = x 表示索引i在x的位置
15         int count;
16         int capacity;
17         // 索引堆中,
18         // 數據之間的比較根據data的大小進行比較,
19         // 但實際操作的是索引
20         void shiftUp(int k) {
21             while (k > 1 && data[indexes[k / 2]] <
22                    data[indexes[k]]) {
23                 swap(indexes[k / 2], indexes[k]);
24                 reverse[indexes[k / 2]] = k / 2;
25                 reverse[indexes[k]] = k;
26                 k /= 2;
27             }
28         }
29         // 索引堆中,
30         // 數據之間的比較根據data的大小進行比較,
31         // 但實際操作的是索引
32         void shiftDown(int k) {
33             while (2 * k <= count) {
34                 int j = 2 * k;
35                 if (j + 1 <= count && data[indexes[j + 1]] > data[indexes[j]])
36                     j += 1;
37                 if (data[indexes[k]] >= data[indexes[j]])
38                     break;
39                 swap(indexes[k], indexes[j]);
40                 reverse[indexes[k]] = k;
41                 reverse[indexes[j]] = j;
42                 k = j;
43             }
44         }
45     public:
46         // 构造函数, 构造一个空的索引堆,
47         // 可容纳capacity个元素
48         IndexMaxHeap(int capacity) {
49             data = new Item[capacity + 1];
50             indexes = new int[capacity + 1];
51             reverse = new int[capacity + 1];
52             for (int i = 0; i <= capacity; i++)
53                 reverse[i] = 0;
54
55             count = 0;
56             this->capacity = capacity;
57         }
58         ~IndexMaxHeap() {
59             delete[] data;
60             delete[] indexes;
61             delete[] reverse;
62         }
63         // 返回索引堆中的元素个数
64         int size() {
65             return count;
66         }
67         // 返回一个布尔值, 表示索引堆中是否为空
68         bool isEmpty() {
69             return count == 0;
70         }
71         // 向最大索引堆中插入一个新的元素,
72         // 新元素的索引为i, 元素为item
73         // 传入的i对用户而言,是从0索引的
74         void insert(int i, Item item) {
75             i += 1;
76             data[i] = item;
77             indexes[count + 1] = i;
78             reverse[i] = count + 1;
79         }
80     };

```

```

73     count++;
74     shiftUp(count);
75 }
76
77 // 从最大索引堆中取出堆顶元素,
78 // 即索引堆中所存储的最大数据
79 Item extractMax() {
80     Item ret = data[indexes[1]];
81     swap(indexes[1], indexes[count]);
82     reverse[indexes[count]] = 0;
83     count--;
84
85     if (count) {
86         reverse[indexes[1]] = 1;
87         shiftDown(1);
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 };

```

3 Graph

3.1 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 };

```

3.2 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 // 路徑查詢
7 template <typename Graph>
8 class Path {
9 private:
10     Graph &G;          // 圖的引用
11     int s;             // 起始點
12     bool *visited;     // 記錄dfs的過程中節點是否被訪問
13     int *from;         // 記錄路徑,
14                       // from[i]表示查找的路徑上i的上一個節點
15 // 圖的深度優先遍歷
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 // 構造函數, 尋路算法,
28 // 尋找圖graph從s點到其他點的路徑
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     // 尋路算法
39     dfs(s);
40 }
41 // 析構函數
42 ~Path() {
43     delete[] visited;
44     delete[] from;
45 }
46 // 查詢從s點到w點是否有路徑
47 bool hasPath(int w) {
48     assert(w >= 0 && w < G.V());
49     return visited[w];
50 }
51 // 查詢從s點到w點的路徑, 存放在vec中
52 void path(int w, vector<int> &vec) {
53     stack<int> s;
54     // 通過from數組逆向查找到從s到w的路徑,
55     // 存放到棧中
56     int p = w;
57     while (p != -1) {
58         s.push(p);
59         p = from[p];
60     }
61     // 從棧中依次取出元素, 獲得順序的從s到w的路徑
62     vec.clear();
63     while (!s.empty()) {
64         vec.push_back(s.top());
65         s.pop();
66     }
67 }
68 // 打印出從s點到w點的路徑
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 };

```

3.3 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 };

```

3.4 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                         visited[i] = true;
45                         from[i] = v;
46                         ord[i] = ord[v] + 1;
47                     }
48             }
49         }
50
51         // 析构函数
52         ~ShortestPath() {
53             delete[] visited;
54             delete[] from;
55             delete[] ord;
56         }
57
58         // 查询从s点到w点是否有路径
59         bool hasPath(int w) {
60             return visited[w];
61         }
62
63         // 查询从s点到w点的路径, 存放在vec中
64         void path(int w, vector<int> &vec) {
65             stack<int> s;
66             // 通过from数组逆向查找到从s到w的路径,
67             // 存放到栈中
68             int p = w;
69             while (p != -1) {
70                 s.push(p);
71                 p = from[p];
72             }
73             // 从栈中依次取出元素, 获得顺序的从s到w的路径
74             vec.clear();
75             while (!s.empty()) {
76                 vec.push_back(s.top());
77                 s.pop();
78             }
79         }
80
81         // 打印出从s点到w点的路径
82         void showPath(int w) {
83             vector<int> vec;
84             path(w, vec);
85             for (int i = 0; i < vec.size(); i++) {
86                 cout << vec[i];
87                 if (i == vec.size() - 1)
88                     cout << endl;
89                 else
90                     cout << " -> ";
91             }
92         }
93
94         // 查看从s点到w点的最短路径长度
95         int length(int w) {
96             assert(w >= 0 && w < G.V());
97             return ord[w];
98         }
99 };

```