

Contents

1	Math	1
1.1	FindPrime	1
2	Heap	1
2.1	IndexMaxHeap	1
3	Graph	2
3.1	Basic	2
3.2	Component	2

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...都被前面處理完了)
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
7 // 最大索引堆
8 template <typename Item>
9 class IndexMaxHeap {
10 private:

```

```

11 Item *data; // 最大索引堆中的數據
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 // 數據之間的比較根據data的大小進行比較,
20 // 但實際操作的是索引
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 // 數據之間的比較根據data的大小進行比較,
32 // 但實際操作的是索引
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 }

```

```

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
90         return ret;
91     }
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
111     // 获取最大索引堆中的堆顶元素的索引
112     int getMaxIndex() {
113         return indexes[1] - 1;
114     }
115
116     // 看索引i所在的位置是否存在元素
117     bool contain(int i) {
118         return reverse[i + 1] != 0;
119     }
120
121     // 获取最大索引堆中索引为i的元素
122     Item getItem(int i) {
123         return data[i + 1];
124     }
125
126     // 将最大索引堆中索引为i的元素修改为newItem
127     void change(int i, Item newItem) {
128         i += 1;
129         data[i] = newItem;
130         shiftUp(reverse[i]);
131         shiftDown(reverse[i]);
132     }
133 };

```

```

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 Graph

3.1 Basic

3.2 Component

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 //visited id 的大小都是Graph的節點數量
5 //範圍從[0, Graph.V()]
6
7 //找連通分量
8 template <typename Graph>
9 class Component {
10 private:
11     Graph &G;

```