Contents

1 Math

1.1 FindPrime

```cpp
#include <bits/stdc++.h>
using namespace std;

//查找[0,2^15]中的所有質數 共有3515

const int MAXN = 32768;  //2^15=32768
bool primes[MAXN];
vector<int> p;  //3515

//質數篩法Sieve of Eratosthenes
inline void findPrimes() {
    for (int i = 0; i < MAXN; i++) {
        primes[i] = true;
    }
    primes[0] = false;
    primes[1] = false;
    for (int i = 4; i < MAXN; i += 2) {
        //將2的倍數全部刪掉(偶數不會是質數)
        primes[i] = false;
    }
    //開始逐個檢查--->小心i*i會有overflow問題--->使用long long
    for (long long i = 3; i < MAXN; i += 2) {
        if (primes[i]) {

            //如果之前還未被刪掉 才做篩法
            for (long long j = i * i; j < MAXN; j += i) {
                //從i*i開始(因為i*2,i*3...都被前面處理完了)
                primes[j] = false;
            }
        }
    }
    //蒐集所有質數
    for (int i = 0; i < MAXN; i++) {
        if (primes[i]) {
            p.emplace_back(i);
        }
    }
}
```

2 Heap

2.1 IndexMaxHeap

```cpp
#include <bits/stdc++.h>
using namespace std;

//data indexes reverse
    陣列邊界都是以[1,capacity]處理--->陣列大小為(capacity+1)

// 最大索引堆
template <typename Item>
class IndexMaxHeap {
   private:
    Item *data;    // 最大索引堆中的数据
    int *indexes;  // 最大索引堆中的索引, indexes[x] = i 表示索引i在x的位置
    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
    int count;
    int capacity;
    // 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
    void shiftUp(int k) {
        while (k > 1 && data[indexes[k / 2]] < data[indexes[k]]) {
            swap(indexes[k / 2], indexes[k]);
            reverse[indexes[k / 2]] = k / 2;
            reverse[indexes[k]] = k;
            k /= 2;
        }
    }
    // 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
    void shiftDown(int k) {
        while (2 * k <= count) {
            int j = 2 * k;
            if (j + 1 <= count && data[indexes[j + 1]] > data[indexes[j]])
                j += 1;

            if (data[indexes[k]] >= data[indexes[j]])
                break;

            swap(indexes[k], indexes[j]);
            reverse[indexes[k]] = k;
            reverse[indexes[j]] = j;
            k = j;
        }
    }

   public:
    // 构造函数, 构造一个空的索引堆,
        可容纳capacity个元素
    IndexMaxHeap(int capacity) {
        data = new Item[capacity + 1];
        indexes = new int[capacity + 1];
        reverse = new int[capacity + 1];
        for (int i = 0; i <= capacity; i++)
            reverse[i] = 0;

        count = 0;
        this->capacity = capacity;
    }
    ~IndexMaxHeap() {
        delete[] data;
        delete[] indexes;
        delete[] reverse;
    }
    // 返回索引堆中的元素个数
    int size() {
        return count;
    }
    // 返回一个布尔值, 表示索引堆中是否为空
    bool isEmpty() {
        return count == 0;
    }
    // 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
    // 传入的i对用户而言,是从0索引的
    void insert(int i, Item item) {
        i += 1;
        data[i] = item;
        indexes[count + 1] = i;
        reverse[i] = count + 1;
        count++;
        shiftUp(count);
```

```
    }

    // 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
    Item extractMax() {
        Item ret = data[indexes[1]];
        swap(indexes[1], indexes[count]);
        reverse[indexes[count]] = 0;
        count--;

        if (count) {
            reverse[indexes[1]] = 1;
            shiftDown(1);
        }

        return ret;
    }

    // 从最大索引堆中取出堆顶元素的索引
    int extractMaxIndex() {
        int ret = indexes[1] - 1;
        swap(indexes[1], indexes[count]);
        reverse[indexes[count]] = 0;
        count--;
        if (count) {
            reverse[indexes[1]] = 1;
            shiftDown(1);
        }
        return ret;
    }
    // 获取最大索引堆中的堆顶元素
    Item getMax() {
        return data[indexes[1]];
    }
    // 获取最大索引堆中的堆顶元素的索引
    int getMaxIndex() {
        return indexes[1] - 1;
    }
    // 看索引i所在的位置是否存在元素
    bool contain(int i) {
        return reverse[i + 1] != 0;
    }
    // 获取最大索引堆中索引为i的元素
    Item getItem(int i) {
        return data[i + 1];
    }

    // 将最大索引堆中索引为i的元素修改为newItem
    void change(int i, Item newItem) {
        i += 1;
        data[i] = newItem;
        shiftUp(reverse[i]);
        shiftDown(reverse[i]);
    }
};
```

```
    bool *visited;
    int ccount = 0;
    int *id;
    void dfs(int v) {
        visited[v] = true;
        id[v] = ccount;
        typename Graph::adjIterator adj(G, v);
        for (int i = adj.begin(); !adj.end(); i =
            adj.next())
            if (!visited[i])
                dfs(i);
    }

  public:
    Component(Graph &graph) : G(graph) {
        visited = new bool[G.V()];
        id = new int[G.V()];
        for (int i = 0; i < G.V(); i++) {
            visited[i] = false;
            id[i] = -1;
        }
        ccount = 0;

        for (int i = 0; i < G.V(); i++)
            if (!visited[i]) {
                dfs(i);
                ccount += 1;
            }
    }
    ~Component() {
        delete[] visited;
        delete[] id;
    }
    int count() {
        return ccount;
    }
    bool isConnected(int v, int w) {
        assert(v >= 0 && v < G.V());
        assert(w >= 0 && w < G.V());
        assert(id[v] != -1 && id[w] != -1);
        return id[v] == id[w];
    }
};
```

3 Graph

3.1 Basic

3.2 Component

```
#include <bits/stdc++.h>
using namespace std;

//visited id 的大小都是Graph的節點數量
    範圍從[0,Graph.V()]

//找連通分量
template <typename Graph>
class Component {
  private:
    Graph &G;
```