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

```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 Graph(No Weight)

2.1 SparseGraph

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

// 稀疏图 - 邻接表
class SparseGraph {
   private:
    int n, m;               // 节点数和边数
    bool directed;          // 是否为有向图
    vector<vector<int>> g;  // 图的具体数据
   public:
    // 构造函数
    SparseGraph(int n, bool directed) {
        assert(n >= 0);
        this->n = n;
        this->m = 0;  // 初始化没有任何边
        this->directed = directed;
        // g初始化为n个空的vector,
            表示每一个g[i]都为空, 即没有任和边
        g = vector<vector<int>>(n, vector<int>());
    }
    ~SparseGraph() {}
    int V() { return n; }  // 返回节点个数
    int E() { return m; }  // 返回边的个数
    // 向图中添加一个边
    void addEdge(int v, int w) {
        g[v].push_back(w);
        if (v != w && !directed)
            g[w].push_back(v);
        m++;
    }
    // 验证图中是否有从v到w的边
    bool hasEdge(int v, int w) {
        for (int i = 0; i < g[v].size(); i++)
            if (g[v][i] == w)
                return true;
        return false;
    }
    // 显示图的信息
    void show() {
        for (int i = 0; i < n; i++) {
            cout << "vertex " << i << ":\t";
            for (int j = 0; j < g[i].size(); j++)
                cout << g[i][j] << "\t";
            cout << endl;
        }
    }
    // 邻边迭代器, 传入一个图和一个顶点,
    // 迭代在这个图中和这个顶点向连的所有顶点
    class adjIterator {
       private:
        SparseGraph &G;  // 图G的引用
        int v;
        int index;

       public:
        // 构造函数
        adjIterator(SparseGraph &graph, int v) :
            G(graph) {
            this->v = v;
            this->index = 0;
        }
        ~adjIterator() {}
        // 返回图G中与顶点v相连接的第一个顶点
        int begin() {
            index = 0;
            if (G.g[v].size())
                return G.g[v][index];
            // 若没有顶点和v相连接, 则返回-1
            return -1;
        }
        // 返回图G中与顶点v相连接的下一个顶点
        int next() {
            index++;
            if (index < G.g[v].size())
                return G.g[v][index];
            // 若没有顶点和v相连接, 则返回-1
            return -1;
        }
```

```
        //
            查看是否已经迭代完了图G中与顶点v相连接的所有顶
        bool end() {
            return index >= G.g[v].size();
        }
    };
};
```

2.2 Path(DFS)

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

//visied from 陣列大小都是圖的頂點數量G.V()

// 路径查询
template <typename Graph>
class Path {
   private:
    Graph &G;         // 图的引用
    int s;            // 起始点
    bool *visited;    // 记录dfs的过程中节点是否被访问
    int *from;        // 记录路径，
        from[i]表示查找的路径上i的上一个节点
    // 图的深度优先遍历
    void dfs(int v) {
        visited[v] = true;
        typename Graph::adjIterator adj(G, v);
        for (int i = adj.begin(); !adj.end(); i =
            adj.next()) {
            if (!visited[i]) {
                from[i] = v;
                dfs(i);
            }
        }
    }

   public:
    // 构造函数，寻路算法，
        寻找图graph从s点到其他点的路径
    Path(Graph &graph, int s) : G(graph) {
        // 算法初始化
        visited = new bool[G.V()];
        from = new int[G.V()];
        for (int i = 0; i < G.V(); i++) {
            visited[i] = false;
            from[i] = -1;
        }
        this->s = s;
        // 寻路算法
        dfs(s);
    }

    // 析构函数
    ~Path() {
        delete[] visited;
        delete[] from;
    }
    // 查询从s点到w点是否有路径
    bool hasPath(int w) {
        assert(w >= 0 && w < G.V());
        return visited[w];
    }
    // 查询从s点到w点的路径，存放在vec中
    void path(int w, vector<int> &vec) {
        stack<int> s;
        // 通过from数组逆向查找到从s到w的路径，
            存放到栈中
        int p = w;
        while (p != -1) {
            s.push(p);
            p = from[p];
        }
        // 从栈中依次取出元素，获得顺序的从s到w的路径
        vec.clear();
        while (!s.empty()) {
            vec.push_back(s.top());
            s.pop();
        }
    }

    // 打印出从s点到w点的路径
    void showPath(int w) {
        vector<int> vec;
        path(w, vec);
        for (int i = 0; i < vec.size(); i++) {
            cout << vec[i];
            if (i == vec.size() - 1)
                cout << endl;
            else
                cout << " -> ";
        }
    }
};
```

2.3 Component

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

//visited id 的大小都是Graph的頂點數量G.V()

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

```
    }
};
```

2.4 ShortestPath(BFS)

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

//visited from ord 陣列大小都是圖的頂點數量G.V()

// 寻找无权图的最短路径
template <typename Graph>
class ShortestPath {
   private:
    Graph &G;        // 图的引用
    int s;           // 起始点
    bool *visited;   // 记录dfs的过程中节点是否被访问
    int *from;       // 记录路径，
        from[i]表示查找的路径上i的上一个节点
    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

   public:
    // 构造函数，
        寻找无权图graph从s点到其他点的最短路径
    ShortestPath(Graph &graph, int s) : G(graph) {
        visited = new bool[graph.V()];
        from = new int[graph.V()];
        ord = new int[graph.V()];
        for (int i = 0; i < graph.V(); i++) {
            visited[i] = false;
            from[i] = -1;
            ord[i] = -1;
        }
        this->s = s;
        // 无向图最短路径算法，
            从s开始广度优先遍历整张图
        queue<int> q;
        q.push(s);
        visited[s] = true;
        ord[s] = 0;
        while (!q.empty()) {
            int v = q.front();
            q.pop();

            typename Graph::adjIterator adj(G, v);
            for (int i = adj.begin(); !adj.end(); i =
                adj.next())
                if (!visited[i]) {
                    q.push(i);
                    visited[i] = true;
                    from[i] = v;
                    ord[i] = ord[v] + 1;
                }
        }
    }

    // 析构函数
    ~ShortestPath() {
        delete[] visited;
        delete[] from;
        delete[] ord;
    }
    // 查询从s点到w点是否有路径
    bool hasPath(int w) {
        return visited[w];
    }
    // 查询从s点到w点的路径，存放在vec中
    void path(int w, vector<int> &vec) {
        stack<int> s;
        // 通过from数组逆向查找到从s到w的路径，
            存放到栈中
        int p = w;
        while (p != -1) {
            s.push(p);
            p = from[p];
        }
        // 从栈中依次取出元素，获得顺序的从s到w的路径
        vec.clear();
        while (!s.empty()) {
            vec.push_back(s.top());
            s.pop();
        }
    }

    // 打印出从s点到w点的路径
    void showPath(int w) {
        vector<int> vec;
        path(w, vec);
        for (int i = 0; i < vec.size(); i++) {
            cout << vec[i];
            if (i == vec.size() - 1)
                cout << endl;
            else
                cout << " -> ";
        }
    }

    // 查看从s点到w点的最短路径长度
    int length(int w) {
        assert(w >= 0 && w < G.V());
        return ord[w];
    }
};
```

3 Graph(Weight)

3.1 SparseGraph

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

// 边
template <typename Weight>
class Edge {
    // 输出边的信息
    friend ostream& operator<<(ostream& os, const
        Edge& e) {
        os << e.a << "-" << e.b << ": " << e.weight;
        return os;
    }

   private:
    int a, b;       // 边的两个端点
    Weight weight;  // 边的权值
   public:
    // 构造函数
    Edge(int a, int b, Weight weight) {
        this->a = a;
        this->b = b;
        this->weight = weight;
    }
    // 空的构造函数，所有的成员变量都取默认值
    Edge() {}

    ~Edge() {}

    int v() { return a; }           // 返回第一个顶点
    int w() { return b; }           // 返回第二个顶点
    Weight wt() { return weight; }  // 返回权值
    // 给定一个顶点，返回另一个顶点
    int other(int x) {
        assert(x == a || x == b);
        return x == a ? b : a;
    }
    // 边的大小比较，是对边的权值的大小比较
    bool operator<(Edge<Weight>& e) {
```

```
        return weight < e.wt();
    }
    bool operator<=(Edge<Weight>& e) {
        return weight <= e.wt();
    }
    bool operator>(Edge<Weight>& e) {
        return weight > e.wt();
    }
    bool operator>=(Edge<Weight>& e) {
        return weight >= e.wt();
    }
    bool operator==(Edge<Weight>& e) {
        return weight == e.wt();
    }
};
```

```
#include <bits/stdc++.h>

#include "Edge.h"
using namespace std;

// 稀疏图 - 邻接表
template <typename Weight>
class SparseGraph {
   private:
    int n, m;                                //
        节点数和边数
    bool directed;                           //
        是否为有向图
    vector<vector<Edge<Weight> *> > g;   //
        图的具体数据
   public:
    // 构造函数
    SparseGraph(int n, bool directed) {
        assert(n >= 0);
        this->n = n;
        this->m = 0;  // 初始化没有任何边
        this->directed = directed;
        // g初始化为n个空的vector,
            表示每一个g[i]都为空, 即没有任和边
        g = vector<vector<Edge<Weight> *> >(n,
            vector<Edge<Weight> *>());
    }
    // 析构函数
    ~SparseGraph() {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < g[i].size(); j++)
                delete g[i][j];
    }
    int V() { return n; }  // 返回节点个数
    int E() { return m; }  // 返回边的个数
    // 向图中添加一个边, 权值为weight
    void addEdge(int v, int w, Weight weight) {
        // 注意, 由于在邻接表的情况,
            查找是否有重边需要遍历整个链表
        // 我们的程序允许重边的出现
        g[v].push_back(new Edge<Weight>(v, w,
            weight));
        if (v != w && !directed)
            g[w].push_back(new Edge<Weight>(w, v,
                weight));
        m++;
    }
    // 验证图中是否有从v到w的边
    bool hasEdge(int v, int w) {
        for (int i = 0; i < g[v].size(); i++)
            if (g[v][i]->other(v) == w)
                return true;
        return false;
    }

    // 显示图的信息
    void show() {
        for (int i = 0; i < n; i++) {
            cout << "vertex " << i << ":\t";
            for (int j = 0; j < g[i].size(); j++)
                cout << "( to:" << g[i][j]->w() <<
                    ",wt:" << g[i][j]->wt() << ")\t";
            cout << endl;
        }
    }
    // 邻边迭代器, 传入一个图和一个顶点,
    // 迭代在这个图中和这个顶点向连的所有边
    class adjIterator {
       private:
        SparseGraph &G;  // 图G的引用
        int v;
        int index;

       public:
        // 构造函数
        adjIterator(SparseGraph &graph, int v) :
            G(graph) {
            this->v = v;
            this->index = 0;
        }
        ~adjIterator() {}
        // 返回图G中与顶点v相连接的第一个边
        Edge<Weight> *begin() {
            index = 0;
            if (G.g[v].size())
                return G.g[v][index];
            // 若没有顶点和v相连接, 则返回NULL
            return NULL;
        }
        // 返回图G中与顶点v相连接的下一个边
        Edge<Weight> *next() {
            index += 1;
            if (index < G.g[v].size())
                return G.g[v][index];
            return NULL;
        }
        //
            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
        bool end() {
            return index >= G.g[v].size();
        }
    };
};
```

3.2 MinimumSpanTree

```
#include <bits/stdc++.h>

#include "Edge.h"
using namespace std;

//陣列大小為傳入的數量n(通常是圖的頂點數量)
// Quick Union + rank + path compression
class UnionFind {
   private:
    int* rank;
    int* parent;  //
        parent[i]表示第i个元素所指向的父节点
    int count;     // 数据个数
   public:
    // 构造函数
    UnionFind(int count) {
        parent = new int[count];
        rank = new int[count];
        this->count = count;
        for (int i = 0; i < count; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }
    // 析构函数
    ~UnionFind() {
        delete[] parent;
        delete[] rank;
```

```
    }
    // 查找过程，查找元素p所对应的集合编号
    // O(h)复杂度，h为树的高度
    int find(int p) {
        // path compression 1
        while (p != parent[p]) {
            parent[p] = parent[parent[p]];
            p = parent[p];
        }
        return p;
    }
    // 查看元素p和元素q是否所属一个集合
    // O(h)复杂度，h为树的高度
    bool isConnected(int p, int q) {
        return find(p) == find(q);
    }
    // 合并元素p和元素q所属的集合
    // O(h)复杂度，h为树的高度
    void unionElements(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if (pRoot == qRoot)
            return;
        // 根据两个元素所在树的元素个数不同判断合并方向
        // 将元素个数少的集合合并到元素个数多的集合上
        if (rank[pRoot] < rank[qRoot]) {
            parent[pRoot] = qRoot;
        } else if (rank[qRoot] < rank[pRoot]) {
            parent[qRoot] = pRoot;
        } else {  // rank[pRoot] == rank[qRoot]
            parent[pRoot] = qRoot;
            rank[qRoot] += 1;  // 此时，我维护rank的值
        }
    }
};

// Kruskal算法
template <typename Graph, typename Weight>
class KruskalMST {
   private:
    vector<Edge<Weight>> mst;  // 最小生成树所包含的所有边
    Weight mstWeight;          // 最小生成树的权值

   public:
    // 构造函数，使用Kruskal算法计算graph的最小生成树
    KruskalMST(Graph& graph) {
        // 将图中的所有边存放到一个最小堆中
        priority_queue<Edge<Weight>> pq(Edge<Weight>,
            vector<Edge<Weight>>,
            greater<Edge<Weight>>);
        for (int i = 0; i < graph.V(); i++) {
            typename Graph::adjIterator adj(graph, i);
            for (Edge<Weight>* e = adj.begin();
                !adj.end(); e = adj.next())
                if (e->v() < e->w())
                    pq.push(*e);
        }

        // 创建一个并查集，来查看已经访问的节点的联通情况
        UnionFind uf = UnionFind(graph.V());
        while (!pq.isEmpty() && mst.size() <
            graph.V() - 1) {
            // 从最小堆中依次从小到大取出所有的边
            Edge<Weight> e = pq.pop();
            pq.pop();
            // 如果该边的两个端点是联通的，说明加入这条边将产生环，扔掉这条边
            if (uf.isConnected(e.v(), e.w()))
                continue;

            // 否则，将这条边添加进最小生成树，同时标记边的两个端点联通
            mst.push_back(e);
            uf.unionElements(e.v(), e.w());
        }

        mstWeight = mst[0].wt();
        for (int i = 1; i < mst.size(); i++)
            mstWeight += mst[i].wt();
    }
    ~KruskalMST() {}
    // 返回最小生成树的所有边
    vector<Edge<Weight>> mstEdges() {
        return mst;
    };
    // 返回最小生成树的权值
    Weight result() {
        return mstWeight;
    };
};
```

3.3 ShortestPath

```
#include <bits/stdc++.h>

#include "Edge.h"

using namespace std;

//data indexes reverse 陣列大小都是capacity大小+1

// 最小索引堆
template <typename Item>
class IndexMinHeap {
   private:
    Item *data;    // 最小索引堆中的数据
    int *indexes;  // 最小索引堆中的索引，indexes[x] = i 表示索引i在x的位置
    int *reverse;  // 最小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]] >
                data[indexes[j + 1]])
                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
    IndexMinHeap(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;
    }
    ~IndexMinHeap() {
        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;
    }
    // 向最小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 index, Item item) {
        index += 1;
        data[index] = item;
        indexes[count + 1] = index;
        reverse[index] = count + 1;
        count++;
        shiftUp(count);
    }

    // 从最小索引堆中取出堆顶元素，
        即索引堆中所存储的最小数据
    Item extractMin() {
        Item ret = data[indexes[1]];
        swap(indexes[1], indexes[count]);
        reverse[indexes[count]] = 0;
        reverse[indexes[1]] = 1;
        count--;
        shiftDown(1);
        return ret;
    }

    // 从最小索引堆中取出堆顶元素的索引
    int extractMinIndex() {
        int ret = indexes[1] - 1;
        swap(indexes[1], indexes[count]);
        reverse[indexes[count]] = 0;
        reverse[indexes[1]] = 1;
        count--;
        shiftDown(1);
        return ret;
    }

    // 获取最小索引堆中的堆顶元素
    Item getMin() {
        return data[indexes[1]];
    }

    // 获取最小索引堆中的堆顶元素的索引
    int getMinIndex() {
        return indexes[1] - 1;
    }

    // 看索引i所在的位置是否存在元素
    bool contain(int index) {
        return reverse[index + 1] != 0;
    }

    // 获取最小索引堆中索引为i的元素
    Item getItem(int index) {
        return data[index + 1];
    }

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

// Dijkstra算法求最短路径
template <typename Graph, typename Weight>
class Dijkstra {
   private:
    Graph &G;                       // 图的引用
    int s;                          // 起始点
    Weight *distTo;                 //
        distTo[i]存储从起始点s到i的最短路径长度
    bool *marked;                   // 标记数组，
        在算法运行过程中标记节点i是否被访问
    vector<Edge<Weight> *> from;    //
        from[i]记录最短路径中，到达i点的边是哪一条
   public:
    // 构造函数，使用Dijkstra算法求最短路径
    Dijkstra(Graph &graph, int s) : G(graph) {
        // 算法初始化
        assert(s >= 0 && s < G.V());
        this->s = s;
        distTo = new Weight[G.V()];
        marked = new bool[G.V()];
        for (int i = 0; i < G.V(); i++) {
            distTo[i] = Weight();
            marked[i] = false;
            from.push_back(NULL);
        }
        //
            使用索引堆记录当前找到的到达每个顶点的最短距离
        IndexMinHeap<Weight> ipq(G.V());
        // 对于其实点s进行初始化
        distTo[s] = Weight();
        from[s] = new Edge<Weight>(s, s, Weight());
        ipq.insert(s, distTo[s]);
        marked[s] = true;
        while (!ipq.isEmpty()) {
            int v = ipq.extractMinIndex();
            // distTo[v]就是s到v的最短距离
            marked[v] = true;
            // 对v的所有相邻节点进行更新
            typename Graph::adjIterator adj(G, v);
            for (Edge<Weight> *e = adj.begin();
                 !adj.end(); e = adj.next()) {
                int w = e->other(v);
                // 如果从s点到w点的最短路径还没有找到
                if (!marked[w]) {
                    // 如果w点以前没有访问过，
                    // 或者访问过，
                        但是通过当前的v点到w点距离更短，
                        则进行更新
                    if (from[w] == NULL || distTo[v]
                        + e->wt() < distTo[w]) {
                        distTo[w] = distTo[v] +
                            e->wt();
                        from[w] = e;
                        if (ipq.contain(w))
                            ipq.change(w, distTo[w]);
                        else
                            ipq.insert(w, distTo[w]);
                    }
                }
            }
        }
    }

    // 析构函数
    ~Dijkstra() {
        delete[] distTo;
```

```
        delete[] marked;
        delete from[0];
    }

    // 返回从s点到w点的最短路径长度
    Weight shortestPathTo(int w) {
        return distTo[w];
    }
    // 判断从s点到w点是否联通
    bool hasPathTo(int w) {
        return marked[w];
    }
    // 寻找从s到w的最短路径，
        将整个路径经过的边存放在vec中
    void shortestPath(int w, vector<Edge<Weight>>
        &vec) {
        // 通过from数组逆向查找到从s到w的路径，
            存放到栈中
        stack<Edge<Weight> *> s;
        Edge<Weight> *e = from[w];
        while (e->v() != this->s) {
            s.push(e);
            e = from[e->v()];
        }
        s.push(e);
        // 从栈中依次取出元素，获得顺序的从s到w的路径
        while (!s.empty()) {
            e = s.top();
            vec.push_back(*e);
            s.pop();
        }
    }

    // 打印出从s点到w点的路径
    void showPath(int w) {
        vector<Edge<Weight>> vec;
        shortestPath(w, vec);
        for (int i = 0; i < vec.size(); i++) {
            cout << vec[i].v() << " -> ";
            if (i == vec.size() - 1)
                cout << vec[i].w() << endl;
        }
    }
};
```

4 Heap

4.1 IndexMaxHeap

```
#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+
//

// 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);
        }
```

```
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 };
```