Contents

Chapter 1

结构

1.1 并查集

```
// 带路径压缩的并查集, 用于动态维护查询等价类
// 图论算法中动态判点集连通常用
// 维护和查询复杂度略大于 O(1)
// 集合元素取值 1..MAXN-1(注意 0 不能用!), 默认不等价

const int MAXN = 100000;

# include <cstring>
# define _run(x) for(; p[t = x]; x = p[x], p[t] = (p[x] ? p[x] : x))
# define _run_both _run(i); _run(j)

class DSet {
```

```
public:
    int p[MAXN],t;

    void init() {
        memset(p, 0, sizeof(p));
    }

    void setFriend(int i, int j) {
        _run_both;
        p[i] = (i == j ? 0 : j);
    }

    bool isFriend(int i, int j) {
        _run_both;
        return i == j && i;
    }
};
```

Chapter 2

图论

2.1 最短路

2.1.1 最短路径

```
//单源最短路径，用于路权相等的情况,dijkstra 优化为 bfs，邻接表形式，复杂度 O(m)
//求出源 s 到所有点的最短路经，传入图的大小 n 和邻接表 list，边权值 len
//返回到各点最短距离 mind[] 和路径 pre[],pre[i] 记录 s 到 i 路径上 i 的父结点,pre[s]=-1
//可更改路权类型，但必须非负且相等！
const int MAXN = 200;
const int INF = 1000000000;

struct Edge {
    int from, to;
    Edge *next;
};

template <class elemType> void dijkstra(int n, const Edge *list[], elemType len, int s, \
elemType *mind, int *pre) {
    Edge *t;
    int i, que[MAXN], f = 0, r = 0, p = 1, L = 1;
    for (i = 0; i<n; i++) {
        mind[i] = INF;
    }
    mind[que[0] = s] = 0;
    pre[s] = -1;
    for (; r <= f; L++, r = f + 1, f = p - 1) {
        for (i = r; i <= f; i++) {
            for (t = list[que[i]]; t; t = t->next) {
                if (mind[t->to] == INF) {
                    mind[que[p++] = t->to] = len * L;
                    pre[t->to] = que[i];
                }
            }
        }
    }
}
```

2.2 生成树

2.2.1 最小生成树

```
//无向图最小生成树,prim 算法, 邻接阵形式, 复杂度 O(n^2)
//返回最小生成树的长度, 传入图的大小 n 和邻接阵 mat, 不相邻点边权 INF
//可更改边权的类型,pre[] 返回树的构造, 用父结点表示, 根节点 (第一个)pre 值为-1
//必须保证图的连通的!
const int MAXN = 200;
const int INF = 1000000000;

template <class elemType>
elemType prim(int n, const elemType mat[][MAXN], int *pre) {
    elemType mind[MAXN], ret = 0;
    int v[MAXN], i, j, k;
    for (i = 0; i<n; i++) {
        mind[i] = INF;
        v[i] = 0;
        pre[i] = -1;
    }
    for (mind[j = 0] = 0; j<n; j++) {
        for (k = -1, i = 0; i<n; i++) {
            if (!v[i] && (k == -1 || mind[i]<mind[k])) {
                k = i;
            }
        }
        v[k] = 1;
        ret += mind[k];
        for (i = 0; i < n; i++) {
            if (!v[i] && mat[k][i] < mind[i]) {
                mind[i] = mat[pre[i] = k][i];
            }
        }
    }
    return ret;
}
```

Chapter 3

应用

3.1 Joseph

```c
// Joseph's Problem
// input: n,m        -- the number of persons, the interval between persons
// output:      -- return the reference of last person

int josephus0(int n, int m) {
    if (n == 2) {
        return (m % 2) ? 2 : 1;
    }
    int v = (m + josephus0(n - 1, m)) % n;
    if (v == 0) {
        v = n;
    }
    return v;
}

int josephus(int n, int m) {
    if (m == 1) {
        return n;
    } else if (n == 1) {
        return 1;
    } else if (m >= n) {
        return josephus0(n, m);
    }
    int l = (n / m) * m;
    int j = josephus(n - (n / m), m);
    if (j <= n - l) {
        return l + j;
    }
    j -= n - l;
    int t = (j / (m - 1)) * m;
    if ((j % (m - 1)) == 0) {
        return t - 1;
    }
    return t + (j % (m - 1));
}
```

3.2 位操作

```
// 遍历一个掩码的所有子集掩码，不包括 0 和其自身
// 传入表示超集的掩码
void iterateSubset(int mask) {
    for(int sub = (mask - 1) & mask; sub > 0; sub = (sub - 1) & mask) {
        int incsub = ~sub & mask; // 递增顺序的子集
        // gogogo
    }
}

// 求一个 32 位整数二进制 1 的位数
// 初始化函数先调用一次

int ones[256];

void initOnes() {
    for (int i = 1; i < 256; ++i)
        ones[i] = ones[i & (i - 1)] + 1;
}

int countOnes(int n) {
    return ones[n & 255] + ones[(n >> 8) & 255] + ones[(n >> 16) & 255] + ones[(n >> 24)\
 & 255];
}

// 求一个 32 位整数二进制 1 的位数的奇偶性
// 偶数返回 0，奇数返回 1
int parityOnes(unsigned n) {
    n ^= n >> 1;
    n ^= n >> 2;
    n ^= n >> 4;
    n ^= n >> 8;
    n ^= n >> 16;
    return n & 1; // n 的第 i 位是原数第 i 位到最左侧位的奇偶性
}
```