万能头文件

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

快速排序

```
void qsort(int l,int r){
    int mid=a[(l+r)/2],i=l,j=r;
    do{while(a[i]<mid)i++;
       while(a[j]>mid)j--;
       if(i<=j){swap(a[i],a[j]);i++;j--;}
    }while(i<=j);
    if(l<j)qsort(l,j);
    if(i<r)qsort(i,r);
}
```

快速幂

```
inline int quickpow(int a,int b){
    int s=1;
    while(b){if(b&1)s=(s*a)%mod;a=(a*a)%mod;b>>=1;}
    return s%mod;
}
```

欧拉筛

```
int prime[maxn];
int visit[maxn];
void Prime(){
    mem(visit,0);
    mem(prime, 0);
    for (int i = 2;i <= maxn; i++) {
        cout<<" i = "<<i<<endl;
        if (!visit[i]) {
            prime[++prime[0]] = i;      //记录素数， 这个prime[0] 相当于
cnt，用来计数
        }
        for (int j = 1; j <=prime[0] && i*prime[j] <= maxn; j++) {
//            cout<<"  j = "<<j<<" prime["<<j<<"]"<<" = "<<prime[j]<<"
i*prime[j] = "<<i*prime[j]<<endl;
            visit[i*prime[j]] = 1;
            if (i % prime[j] == 0) {
                break;
```

```
            }
        }
    }
}
```

并查集

```
inline int fd(int k){
    if(f[k]==k)return k;
    return f[k]=fd(f[k]);
}inline void mr(int v,int u){
    int t1=fd(v),t2=fd(u);
    if(t1!=t2)f[t2]=t1;
}
```

离散化

```
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素
// 二分求出 x 对应的离散化的值
int find(int x) // 找到第一个大于等于 x 的位置
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到 1, 2, ...n
}
```

KMP

```
// s[]是长文本，p[]是模式串，n 是 s 的长度，m 是 p 的长度
求模式串的 Next 数组：
for (int i = 2, j = 0; i <= m; i ++ )
{
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
    ne[i] = j;
}
// 匹配
for (int i = 1, j = 0; i <= n; i ++ )
```

```
{
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

一般哈希

1) 拉链法

```
int h[N], e[N], ne[N], idx;
// 向哈希表中插入一个数
void insert(int x)
{
    int k = (x % N + N) % N;
    e[idx] = x;
    ne[idx] = h[k];
    h[k] = idx ++ ;
}
// 在哈希表中查询某个数是否存在
bool find(int x)
{
    int k = (x % N + N) % N;
    for (int i = h[k]; i != -1; i = ne[i])
        if (e[i] == x)
            return true;
    return false;
}
```

(2) 开放寻址法

```
int h[N];
// 如果 x 在哈希表中，返回 x 的下标；如果 x 不在哈希表中，返回 x 应该插入的位置
int find(int x)
{
    int t = (x % N + N) % N;
    while (h[t] != null && h[t] != x)
    {
        t ++ ;
        if (t == N) t = 0;
    }
    return t;
}
```

字符串哈希

```
typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k]存储字符串前 k 个字母的哈希值, p[k]存储  P^k mod 2^64
// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}
// 计算子串  str[l ~ r]  的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

邻接表存图

```
// 对于每个点 k，开一个单链表，存储 k 所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;
// 添加一条边 a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}
// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

拓扑排序

```
bool topsort()
{
    int hh = 0, tt = -1;
    // d[i]  存储点 i 的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;
    while (hh <= tt)
    {
        int t = q[hh ++ ];
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
```

```
    }
    // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
    return tt == n - 1;
}
```

堆优化 Dij

```
typedef pair<int, int> PII;
int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;           // 邻接表存储所有边
int dist[N];            // 存储所有点到 1 号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定
// 求 1 号点到 n 号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});          // first 存储距离，second 存储节点编号
    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();
        int ver = t.second, distance = t.first;
        if (st[ver]) continue;
        st[ver] = true;
        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }
    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

朴素 prim 求最小生成树（n 方+m）

```
int n;      // n 表示点数
int g[N][N];            // 邻接矩阵，存储所有边
int dist[N];            // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中
```

```
// 如果图不连通，则返回 INF(值是 0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);
    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;
        if (i && dist[t] == INF) return INF;
        if (i) res += dist[t];
        st[t] = true;
        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }
    return res;
}
```

Kruskal 求最小生成树（mlogm）

```
int n, m;           // n 是点数，m 是边数
int p[N];           // 并查集的父节点数组
struct Edge         // 存储边
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];
int find(int x)         // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}
int kruskal()
{
    sort(edges, edges + m);
    for (int i = 1; i <= n; i ++ ) p[i] = i;      // 初始化并查集
    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
```

```
            a = find(a), b = find(b);
            if (a != b)       // 如果两个连通块不连通，则将这两个连通块合并
            {
                p[a] = b;
                res += w;
                cnt ++ ;
            }
        }
        if (cnt < n - 1) return INF;
        return res;
}
```

染色法判别二分图

```
int n;          // n 表示点数
int h[N], e[M], ne[M], idx;       // 邻接表存储图
int color[N];           // 表示每个点的颜色，-1 表示未染色，0 表示白色，1 表示黑色
// 参数：u 表示当前节点，c 表示当前点的颜色
bool dfs(int u, int c)
{
        color[u] = c;
        for (int i = h[u]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (color[j] == -1)
            {
                if (!dfs(j, !c)) return false;
            }
            else if (color[j] == c) return false;
        }
        return true;
}
bool check()
{
        memset(color, -1, sizeof color);
        bool flag = true;
        for (int i = 1; i <= n; i ++ )
            if (color[i] == -1)
                if (!dfs(i, 0))
                {
                    flag = false;
                    break;
                }
        return flag;
}
```