Bellman-Ford

时间复杂度O(nm)

```
int n, m;       // n表示点数，m表示边数
int dist[N];        // dist[x]存储1到x的最短路距离

struct Edge     // 边，a表示出点，b表示入点，w表示边的权重
{
    int a, b, w;
}edges[M];

// 求1到n的最短路距离，如果无法从1走到n，则返回-1。
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
    for (int i = 0; i < n; i ++ )
    {
        for (int j = 0; j < m; j ++ )
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if (dist[b] > dist[a] + w)
                dist[b] = dist[a] + w;
        }
    }

    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

//字符串hash

```
// p = 131 或133
f[i] = s[i] - 'a' + f[i - 1]*p 表示0到i字符串的hash值

f[j] - f[i ] * p^j 表示字符串s[i]到s[j]的hash值
```

#

(a/b) % mod == a * b1 % mod

b1 = b^(mod - 2)

//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
        }
    }
```

Kruskal

```
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;
}
```

// LIS(最长升序子序列)

```
// lower_bound(q, q + len, val); 返回非递减序列q到q + len的第一个大于等于val的元素的迭代器
// upper_bound(q, q + len, val); 返回非递减序列q到q + len的第一个大于val的元素的迭代器
int n, a[1005], idx ;
int lis(){
    for(int i = 0; i < n; i ++)
    if(idx == 0 ||a[i] > a[idx - 1]) a[idx++] = a[i];
    else *lower_bound(a , a + idx , a[i]) = a[i];
    return idx;
}
```

// 手动二分版

```
int lis(){
    for(int i = 0; i < n; i ++)
    if(idx == 0 ||a[i] > a[idx - 1]) a[idx++] = a[i];
    else {
        int l = 0, r = idx;
        while(l < r){
            int mid = l + r >> 1;
            if(a[mid] >= a[i]) r = mid;
            else l = mid + 1;
        }
        a[r] = a[i];
    }
    return idx;
}
```

spfa

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1
int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

    while (q.size())
    {
```

```
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])     // 如果队列中已存在j，则不需要将j重复插入
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

堆优化Dijkstra

时间复杂度 O(mlogn)

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx = 1;       // 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

void add(int u, int v, int d){
    e[idx] = v, w[idx] = d, ne[idx] = h[u], h[u] = idx++;
}
// 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 ; 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];
}
```

// 归并排序

```
int a[N], tmp[N];
void m_sort(int l, int r){
    if(l >= r) return;
    int mid = l + r >> 1;
    m_sort(l, mid), m_sort(mid + 1, r);
    int i = l, j = mid + 1, k = 0;
    while(i <= mid && j <= r){
        if(a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];
    }
    while(i<=mid) tmp[k++] = a[i++];
    while(j <= r) tmp[k++] = a[j++];
    for(int i = 0; i < k; i ++) a[l + i] = tmp[i];
}
```

// 快速排序quick_sort

```
int q[N]
void q_sort(int q[], int l, int r){
    if(l >= r) return;
    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while(i < j){
        do i++; while(q[i] < x);
        do j--; while(q[j] > x);
        if(i < j) swap(q[i], q[j]);
    }
    q_sort(q, l, j), q_sort(q, j + 1, r);
}
```

扩展欧几里得

```
int exgcd(int a, int b, int &x, int &y){
    //cout << x << " "<< y << endl ;
    if(!b){
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
```

ax + by = gcd(a, b)

==> a(x0 + b/d*k) + b(y0 + a/d*k) = gcd(a, b)

==>minx = (x0%(b/d) +b/d) %(b/d), miny = (y0%(a/d) +a/d) %(a/d)

//链式前向星（带边权）

```
// idx离散化边； h[i] 存以i为起点的边的离散化编号； e[i]存编号为i的边，e[i].to表示该边的终点，e[i].w表示边权，e[i].ne表示上一条起点相同的边。
struct edge{
    int to, ne, w;
}e[N];
int idx, h[505];

void add(int u, int v, int w){
    e[idx].to = v, e[idx].w = w, e[idx].ne = h[u], h[u] = idx++;
}

//初始化 memset(h, -1, sizeof h);
```

//欧拉线性筛

```
bool visit[N];            //visit标记合数
int prime[N], cnt;     //prime存所有质数
void get_prime(int n){
    for(int i = 2; i <= n; i ++){
        if( !visit[i] ) prime[cnt++] = i;
            for( int j = 0 ; prime[j] <= n/i ; j ++ ){
                visit[prime[j] * i] = 1;        //标记质数的i倍的合数
                if(i % prime[j] == 0) break;    //线性优化关键
            }
    }
}
```

//朴素线段树

```
struct node{
    int l, r, val;
}tr[4*N];
int q[N];
void pushup(int u){
    tr[u].val = tr[u << 1].val + tr[u << 1 | 1].val;
}
void build(int u, int l, int r){
```

```
    if(l == r) tr[u] = {l , r, q[l]};
    else{
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        tr[u] = {l, r};
        pushup(u);
    }
}
void modify(int u, int x, int v){
    if(tr[u].l == tr[u].r) tr[u].val += v;
    else{
        int mid = tr[u].l + tr[u].r >> 1;
        if(x <= mid) modify(u << 1, x, v);
        else modify(u << 1 | 1, x, v);
        pushup(u);
    }
}
int query(int u, int l, int r){
    if(tr[u].l >= l && tr[u].r <= r) return tr[u].val;
    int sum = 0;
    int mid = tr[u].l + tr[u].r >> 1;
    if(l <= mid) sum += query(u << 1, l, r);
    if(r > mid) sum += query(u << 1 | 1, l, r);
    return sum;
}
```

//树状数组

```
int tr[N];
inline int lowbit(int x){ return x & -x; }
void add(int x, int v){      //单点更新
    for(int i = x; i <= n; i += lowbit(i)) tr[i] += v;
}
int query(int x){        //区间查询
    int sum = 0;
    for(int i = x; i ; i -= lowbit(i)) sum += tr[i];
    return sum;
}
```

数字离散化

```
vector<int>a(N);
sort(a.begin(), a.end());
a.erase(unique(a.begin(), a.end()), a.end());    //去重
```