基础算法

快速排序算法模板

先排序再递归，每次确定一个元素的位置，并且根据该元素的值将左右分块，左边都小于x，右边都大于等于x

时间复杂度 NlogN

已知应用:

普通排序

寻找第K大/小的数： j - l + 1 >= k 即第k个数在左侧，否则就在右侧：K变为 k - (j - l + 1)，这种方式使得时间复杂度降为logN

```
void quick_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)
    {
        while(q[ ++ i] < x) ;
        while(q[ -- j] > x]) ;
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}
```

归并排序算法模板

先递归再排序，需要借助中间数组存储已排序序列

时间复杂度 NlogN

已知应用:

普通排序

逆序对：在排序的过程中寻找逆序对，逆序对只存在于归并过程中，一个序列中不会存在逆序对，因为两个序列合并成一个序列时已经有序， res += mid - l + 1

```
void merge_sort(int q[], int l, int r)
{
    if (l == r) return;

    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];
```

```
    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

整数二分算法模板

使用条件：序列有序

时间复杂度：logN

注意事项：、

mid = l + r >> 1，寻找的是第一个大于等于x的数

mid = l + r + 1 >> 1，寻找的是第一个小于等于x的数

对于二分一定要灵活判断，只要是需要将区间一直分为两块，就可以注意

已知应用：

数的查找 --- x在序列的最左端位置 和 最右端位置 / x在序列的长度

```
// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

浮点数二分算法模板

注意事项：

一定要是double类型，不能是float类型，否则会超时

```
bool check(double x) {/* ... */} // 检查x是否满足某种性质

double bsearch_3(double l, double r)
{
    const double eps = 1e-6;   // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

高精度加法

高精度减法、除法基本上很少用，高精度加法、乘法在某些题当中会用到，比如说输入范围或者输出范围可能会爆int，此时就需要通过字符串来算值等，具体根据实际情况来，如剑指offer67.将字符串转为整数

```
// C = A + B, A >= 0, B >= 0
vector<int> add(vector<int> &A, vector<int> &B)
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }

    if (t) C.push_back(t);
    return C;
}
```

高精度减法

```
// C = A - B, 满足A >= B, A >= 0, B >= 0
vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i ++ )
    {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
```

```
}
```

高精度乘法

```
// C = A * b, A >= 0, b >= 0
vector<int> mul(vector<int> &A, int b)
{
    vector<int> C;

    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ )
    {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

高精度除法

```
// A / b = C ... r, A >= 0, b > 0
vector<int> div(vector<int> &A, int b, int &r)
{
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- )
    {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

一维前缀和

```
S[i] = a[1] + a[2] + ... a[i] --- S[i] = S[i - 1] + a[i] 求前i个数的和
a[l] + ... + a[r] = S[r] - S[l - 1] 求[l, r]的和
```

二维前缀和

```
S[i, j] = 第i行j列格子左上部分所有元素的和 = S[i - 1, j] + S[i, j - 1] - S[i - 1, j - 1] + a[i, j]
以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为:
S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1] --- 受伤的只有1
```

一维差分

```
给区间[l, r]中的每个数加上c：a[l] += c, a[r + 1] -= c
a[i] = s[i] - s[i - 1] //输入前缀和的同时求出单值a[i]
```

二维差分

```
给以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵中的所有元素加上c：
a[x1, y1] += c, a[x2 + 1, y1] -= c, a[x1, y2 + 1] -= c, a[x2 + 1, y2 + 1] += c//
在输入前缀和S[i]的同时用该公式求出单项a[i]
```

位运算

已知应用：

公式1常用与状态压缩dp问题，判断第k位是0或1(0 1表示两种不同状态)

公式2用于统计二进制中1的个数

```
求n的第k位数字: n >> k & 1 //公式1
返回n的最后一位1：lowbit(n) = n & -n //公式2
```

双指针算法

常见问题分类：

(1) 一个序列，两个指针从 同一侧 / 从不同侧 开始移动

同一侧：最长连续递增子序列 变化：寻找链表倒数第K个点 --- 一个指针先走K步，然后再一起走

不同侧：二数之和、三数之和

(2) 两个序列，两个指针从 同一侧 / 从不同侧 开始移动

同一侧：二路归并排序 变化：链表第一个相交的点 --- 指针x，y一起走，短的先走完，然后重新指向长链表头，长的走完重新指向短链表头，从而使得其新的开 始离交汇点距离相同

不同侧：寻找目标和

```
for (int i = 0, j = 0; i < n; i ++ )
{
    while (j < i && check(i, j)) j ++ ;

    // 具体问题的逻辑
}
```

离散化

思想：

杂乱的书籍随意放在极大的空间中，即实际上很少的数据量却占据了超出范围的内存空间，所以将杂乱的书籍整理以便用有限的空间存储下来

```
vector<int> alls; // 存储所有待离散化的值 --- 将所有需要整理的书籍记下来
sort(alls.begin(), alls.end()); // 将所有值排序 --- 整体书籍
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
}
```

区间合并

```
// 将所有存在交集的区间合并
void merge(vector<PII> &segs)
{
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;
    for (auto seg : segs)
        if (ed < seg.first)// 区间之间不存在交集
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);// 存在交集取最右端

    if (st != -2e9) res.push_back({st, ed});//将最后一个合并的区间放入

    segs = res;
}
```

数据结构

单链表

```
//数组模拟链表
// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;

// 初始化
void init()
{
    head = -1;
    idx = 0;
}

// 在链表头插入一个数a
void add(int a)
{
    e[idx] = a, ne[idx] = head, head = idx ++ ;
}
```

```
// 将头结点删除，需要保证头结点存在
void remove()
{
    head = ne[head];
}
```

双链表

```
// e[]表示节点的值，l[]表示节点的左指针，r[]表示节点的右指针，idx表示当前用到了哪个节点
int e[N], l[N], r[N], idx;

// 初始化
void init()
{
    //0是左端点，1是右端点
    r[0] = 1, l[1] = 0;
    idx = 2;
}

// 在节点a的右边插入一个数x
void insert(int a, int x)
{
    e[idx] = x;
    l[idx] = a, r[idx] = r[a];
    l[r[a]] = idx, r[a] = idx ++ ;
}

// 删除节点a
void remove(int a)
{
    l[r[a]] = l[a];
    r[l[a]] = r[a];
}
```

栈

```
// tt表示栈顶
int st[N], tt = -1;

// 向栈顶插入一个数
stk[ ++ tt] = x;

// 从栈顶弹出一个数
tt -- ;

// 栈顶的值
stk[tt];

// 判断栈是否为空
if (tt == -1)
{

}
```

队列

已知应用:

　滑动窗口等需要求出当前某个序列最大最小值问题

```
1.普通队列
// hh 表示队头，tt表示队尾
int q[N], hh = 0, tt = -1;

// 向队尾插入一个数
q[ ++ tt] = x;

// 从队头弹出一个数
hh ++ ;

// 队头的值
q[hh];

// 判断队列是否为空
if (hh <= tt)
{

}
```

```
1.循环队列 --- 即循环使用有效空间
// hh 表示队头，tt表示队尾的后一个位置
int q[N], hh = 0, tt = 0;

// 向队尾插入一个数
q[tt ++ ] = x;
if (tt == N) tt = 0;

// 从队头弹出一个数
hh ++ ;
if (hh == N) hh = 0;

// 队头的值
q[hh];

// 判断队列是否为空
if (hh != tt)
{

}
```

单调栈

常见模型:

　找出每个数左边/右边离它最近的比它大/小的数

```
int tt = 0;
for (int i = 1; i <= n; i ++ )
{
    while (tt && check(stk[tt], i)) tt -- ;
    stk[ ++ tt] = i;
}
```

单调队列

常见模型：

找出滑动窗口中的最大值/最小值

找出当前序列的最大/最小值 --- 带min/max函数的队列

```
int hh = 0, tt = -1;
for (int i = 0; i < n; i ++ )
{
    while (hh <= tt && check_out(q[hh])) hh ++ ;  // 判断队头是否滑出窗口
    while (hh <= tt && check(q[tt], i)) tt -- ;
    q[ ++ tt] = i;
}
```

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
    }
}
```

Trie树

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
```

```
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}

Trie的变化应用 --- 最大异或对
void add(int x)
{
    int t = 0;
    for(int i = 30; ~i; i -- )
    {
        if(!trie[t][x >> i & 1]) trie[t][x >> i & 1] = ++ idx;
        t = trie[t][x >> i & 1];
    }
}

int query(int x)
{
    int t = 0, sum = 0;
    for(int i = 30; ~i; i -- )
    {
        if(trie[t][!(x >> i & 1)]) sum += 1 << i, t = trie[t][!(x >> i & 1)];
        else t = trie[t][x >> i & 1];
    }
    return sum;
}
```

并查集

```
(1)朴素并查集：

    int p[N]; //存储每个点的祖宗节点

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ ) p[i] = i;

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
```

```
(2)维护size的并查集：

    int p[N], size[N];
    //p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
        p[i] = i;
        size[i] = 1;
    }

    // 合并a和b所在的两个集合：
    size[find(b)] += size[find(a)];
    p[find(a)] = find(b);


(3)维护到祖宗节点距离的并查集：

    int p[N], d[N];
    //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x)
        {
            int u = find(p[x]);
            d[x] += d[p[x]];
            p[x] = u;
        }
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
        p[i] = i;
        d[i] = 0;
    }

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
    d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

堆

```
// h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1
// ph[k]存储第k个插入的点在堆中的位置
```

```
// hp[k]存储堆中下标是k的点是第几个插入的
int h[N], ph[N], hp[N], size;

// 交换两个点，及其映射关系
void heap_swap(int a, int b)
{
    swap(ph[hp[a]],ph[hp[b]]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u)
{
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)
    {
        heap_swap(u, t);
        down(t);
    }
}

void up(int u)
{
    while (u / 2 && h[u] < h[u / 2])
    {
        heap_swap(u, u / 2);
        u >>= 1;
    }
}

// O(n)建堆
for (int i = n / 2; i; i -- ) down(i);
```

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 --- h数组的长度选择第一个超出数据范围的质数
    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] != -2e9 && h[t] != x)
        {
            t ++ ;
            if (t == N) t = 0;
        }
        return t;
    }
```

字符串哈希

```
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

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;//哈希值不取0，因为A和AA都是0，即哈希值映射不唯一
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];//hash值，左边的字符串是P进制的高位，后面是低位，即字符串从短变长后，新进的是低位，原有的低位进到高位
    //从而保证了l - 1位进r - l + 1位与r的哈希值位数相等且高位数也相等
    p[i] = p[i - 1] * P;//记录当前hash值得指数
}

// 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];//h[l - 1] 得到的是低位1 - l - 1的哈希值，h[r]得到的是1-r的哈希值
    //低位哈希值到高位哈希值的位数差距是r - l + 1
}
```

图论

树与图的存储

邻接矩阵 --- g[a, b]存储的是a->b的边，适用于稠密图，即边m与点n不在同一个数量级，起码是平方差距

邻接表 --- 适用于稀疏图，即边m和点n处于同一个数量级

```
// 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;//W[N]

// 添加一条边a->b
void add(int a, int b// int w)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;变化：存储a->b边的权值 W[idx] = w;
}

// 初始化
idx = 0;
memset(h, -1, sizeof h);// 该函数只适用于赋值-1， 0x3f，true，false有限情况
```

树与图的遍历

时间复杂度 O(n + m)

1.深度优先遍历 --- 常用来找路径 ， 但是题目不一定明确指出是要你找路径，而是有许多变化，如找符合条件的字符串、树的深度等

注意点：

需要根据实际情况来决定是否回溯，找路径就需要回溯，只是发生交换、比值就不需要回溯

要额外注意返回值的情况 --- (1)返回给上层的是什么 (2)下层递归的修改能否作用到上一次递归

```
1.数组存储数据形式
int dfs(int u)
{
    st[u] = true; // st[u] 表示点u已经被遍历过

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}
2.给定一个结点
void dfs(TreeNode* root, int sum, vector<int> path)
    {
        if(!root) return ;
        sum -= root->val;
        path.push_back(root->val);
        if(!sum && !root->left && !root->right)
        {
            res.push_back(path);
            return ;
        }
        if(root->left) dfs(root->left, sum, path);
        if(root->right) dfs(root->right, sum, path);
        sum -= path[path.size() - 1];//回溯
        path.pop_back();//回溯
    }
3.用数组标记已访问
void dfs(int x){//递归参数
    if(x == n){//递归终止条件
        for(int i = 0; i < n; i ++ ){
            cout << path[i] << " ";
```

```
        }
        puts("");
        return ;
    }
    for(int i = 0; i < n; i ++ ){//单层搜索逻辑
        if(!is[i]){
            path[x] = i + 1;
            is[i] = true;
            dfs(x + 1);
            is[i] = false;//回溯
        }else continue;
    }
}
4.未运算标记
void dfs(int x, int state){
    if(x == n){
        for(int i = 0; i < n; i ++ ){
            cout << path[i] << " ";
        }
        puts("");
        return ;

    }

    for(int i = 0; i < n; i ++ ){
        if(!(state >> i & 1)){//判定第i位是否用过
            path[x] = i + 1;
            dfs(x + 1, state + (1 << i));//标记第i位已经使用
        }
    }
    return ;
}
```

宽度优先遍历

已知应用:

树的宽度、深度

树的层次遍历

树的最左侧/最右侧点

```
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

while (q.size())
{
    int t = q.front();
    q.pop();

    for (int i = h[t]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true; // 表示点j已经被遍历过
```

```
            q.push(j);
        }
    }
}
```

拓扑排序

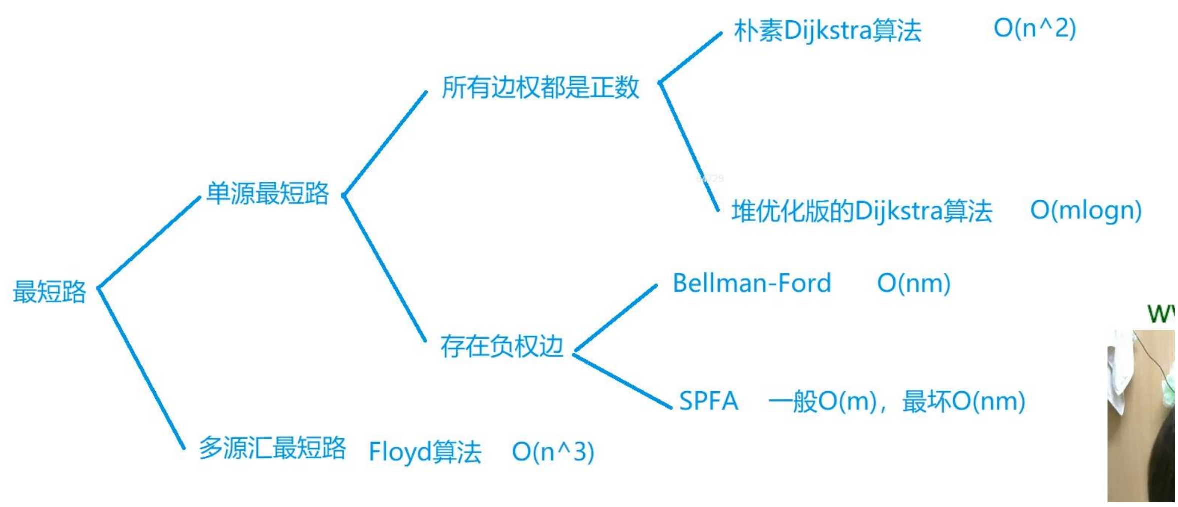
已知应用:

求拓扑序列

判定有向图是否有环

```
bool topsort()//变化：求拓扑序列，出队的顺序就是一个拓扑序列，用一个数组存储下来即可
{
    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;
}
```

最短路算法总结



朴素Dijkstra算法思路:

1.逐步确定每个点到起始点的最短距离(一重for循环)

2.找到当前未确定距离的最小值，并且将其加入到确定范围内(一重for循环)

3.用新加入的点更新最短距离(一重for循环)

堆优化版本dijkstra

优化点：第二步通过最小堆使得时间复杂度从O(N)变为O(1)，更新距离时时间复杂度为logN

Bellman算法优化：

优化点：从dijkstra算法的**由点找边**变为Bellman算法的**直接遍历边**来更新值，其用来处理限制边数的相关问题，整体过程如下

1.限制要走的边数

2.备份上一轮更新后的距离

2.遍历所有边，并且用走的边来更新距离，但是要注意的是更新距离的时候要使用备份距离数据

spfa算法优化：

bellman算法是**遍历所有边**，无论该边之前的距离是否发生改变，spfa算法就根据这一点进行优化，**只有前面的边的最短距离发生变化，其后面跟着的边最短距离才可能发生变化**

spfa算法的**st数组不再像dijkstra算法用于标记该点的最短距离是否已经确定**，而是**标记该点的距离是否发生了变化**，发生了变化就要入队用其更新后面点的最短距离

Floyd算法：

无脑暴力循环，只要注意中转点在第一重循环即可

朴素Dijkstra算法

适应情况：稠密图，正权边

时间复杂度 O(n^2 + m)

```
int dijkst(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    for(int i = 1; i <= n; i ++ ){//寻找所有点到起点的最短距离
        int t = -1;
        for(int j = 1; j <= n; j ++ ){//找到未确定且距离最小的点
            if(!st[j] && (t == -1 || dist[t] > dist[j]))
            t = j;
        }
        st[t] = true;//将该点确定
        for(int j = 1; j <= n; j ++ ){//用该点距离更新其他点
            dist[j] = min(dist[j], dist[t] + g[t][j]);
        }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

堆优化版dijkstra

适应情况：稀疏图，正权边

时间复杂度 O(mlongn) --- 堆每次更新值时间复杂度是logn，而通过邻接表来存，

每次只遍历与该点相连的边，所以总的遍历次数是m，故时间复杂度是mlogn

```
int dijkstra(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>,greater<PII>> heap;
    heap.push({0, 1});
    while(heap.size()){//第一步遍历
        auto t = heap.top();//第二步①找出未确定的距离最小的点
        heap.pop();
        int dis = t.first, ver = t.second;
        if(st[ver]) continue;
        st[ver] = true;//第二步②将该最短距离确定下来
        for(int i = he[ver]; i != -1; i = ne[i]){//第三步 更新dist数组
            int j = e[i];
            if(dist[j] > dist[ver] + w[i]){
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j], j});
                //此处会产生冗余，对于产生新的最短距离的点，其{旧值距离，点}会成为冗余数据，
                //下沉到堆得下半部分
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Bellman-Ford算法

适用于：限制边数且存在负权，可以用来判定是否存在负权回路，但是一般用spfa算法

时间复杂度 O(nm) --- k条边 + 每次遍历更新所有边

```
void bellman_ford(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    for(int i = 0; i < k; i ++ ){//限制只能走K条边
        memcpy(last, dist, sizeof dist);//对边更新之前先备份，避免连锁反应
        for(int j = 0; j < m; j ++ ){
            int a = edgs[j].a, b = edgs[j].b, w = edgs[j].w;
            dist[b] = min(dist[b], last[a] + w);//用上一轮更新得dist值（backup）来进
行比较更新
        }
    }
}
```

spfa 算法（队列优化的Bellman-Ford算法)

适用于：负权边，可以用来求最短距离、是否存在负权回路，可以用于替代dijkstra算法，如果无法ac再换算法

时间复杂度 平均情况下 O(m)O(m)，最坏情况下 O(nm)

```
1.求最短距离
int spfa(){
    memset(dist, 0x3f, sizeof dist);
    queue<int> que;
    dist[1] = 0;
```

```
    st[1] = true;
    que.push(1);
    while(que.size()){
        int t = que.front();
        que.pop();
        st[t] = false;
        for(int i = he[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]){
                    st[j] = true;
                    que.push(j);
                }
            }
        }
    }
    return dist[n];
}
2.是否存在负权回路
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], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。

    queue<int> q;
    for (int i = 1; i <= n; i ++ )//是求是否存在负权回路，而不是从起点到终点，所以必须将所有点加入进来
    {                              //其实模拟一遍就会发现因为dist数组没有进行初始化，那么第一次产生距离更新一定是负权边开始
        q.push(i);
        st[i] = 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];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n) return true;       // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环
```

```
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

floyd算法

```
初始化：
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )//无脑通过中间点暴力循环
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

最小生成树两大算法 --- prim算法 和 kruskal

朴素版prim算法

思想：从某个点出发，并将这个点纳入集合，每次找到**离集合最近的点**，纳入集合，并且**更新**未纳入集合的**点到集合的距离**

存储方式：领接矩阵

整体过程和dijkstra一模一样，只是距离的含义产生了变化

```
int prim(){
    dist[1] = 0;
    int res = 0;
    for(int i = 1; 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(dist[t] == 0x3f3f3f3f) return 0x3f3f3f3f;//最短距离都是无穷，说明不连通
        res += dist[t];
        st[t] = true;

        for(int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }
    return res;//求出最小生成树的最短距离
}
```

Kruskal算法

思想：排序，每次取未纳入集合并且最小的边，将其纳入集合中，直到所有点都纳入到集合中

存储方式：邻接表

```
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 --- 用于将边纳入集合
{
    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;//cnt统计纳入的边数
    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;//n个点最小纳入次数是n - 1，只有这样才保证树是连通的
    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)//未染色
        {
            if (!dfs(j, !c)) return false;//与u点连通的j点染色失败，即颜色冲突
        }
        else if (color[j] == c) return false;//已经染色并且与u点颜色相同
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )//每个点尝试染色
        if (color[i] == -1)//没有染色即开始尝试染色
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

匈牙利算法

```
int n1, n2;     // n1表示第一个集合中的点数，n2表示第二个集合中的点数
int h[N], e[M], ne[M], idx;     // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
int match[N];       // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))//该女子没有匹配的人 或者 该女子喜欢的男生有备胎所以将其让出来
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}
```

```
// 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);//不管女生是否有喜欢的人，我都要尝试一次，只有彻底失败我才认
    if (find(i)) res ++ ;
}
```