2024暑假

一、数据结构

1. 指针

- 若指针p1，p2指向同一个数组，p2 - p1表示两者之间的元素个数

2. 线性表

1. 顺序表

> 数组(date)和长度(length)

1. 在顺序表中删除值为x的元素（时间复杂度为O(x)）
   - 直接在此表中将非x元素进行重构
   - 一边扫描并更新x的个数(k),一边让非x元素向前移动k个位置
2. 顺序表第一个元素为基准，小的移到前面，大的移到后面（时间复杂度为O(x)）
   - 先将基数存起来，从后往前找小的，从前往后找大的，两两交换，最后指向相同时，与基数位置交换
   - 将基数存起来，从后往前找小的，移至基数位置，原位置空缺，再从前往后循环往返，将每个符合要求的数移到空缺处，最后将基数移到空缺处（优解）

```
while( i < j)
    {
        while( i < j && L->data[j] <= a)//从后往前找小的
        {
            j--;
            L->data[i] = L->data[j];
        }
        while( i < j && L->data[i] > a)//从前往后找大的
        {
            i++;
            L->data[j] = L->data[i];
        }
        if( i ==j )//最后将基数移到空缺处
        {
            L->data[i] = a;
        }
    }
```

2. 链式存储结构

> 存储密度：节点数据占用内存 / 节点占用内存

存储密度小，删除和插入元素时间复杂度为O(1)

1. 单链表

2. 双链表

3. 循环链表（单链表和双链表）

3. 栈

1. 顺序存储

```
struct Stack
{
    ElemType data[maxsize]
    int top = -1//栈顶指针
}
```

共享栈

> 有两个栈顶指针

2. 链式存储

- 栈空：s->next=NULL(s为头结点)
- 进栈：将包含e的节点插入到头结点之后

3. 应用

1. 逆波兰表达式
    - 将中缀表达式转化为逆波兰表达式

    ```
    //转化为逆波兰表达式
    string trans(string& s)
    {
        string operand;
        stack<char> Operator;
        int flag = 0;//记录括号优先级
        for (const auto& e : s)//一个一个遍历
        {
            if (e == '(')
            {
                Operator.push(e);
                flag = 1;
                continue;
            }
            if (e == ')')
            {
                flag = 0;
                while (Operator.top() != '(')
                {
                    operand.push_back(Operator.top());
                    Operator.pop();
                }
                Operator.pop();
                continue;
            }
            //操作符
    ```

```
        if (e == '+' || e == '-' || e == '*' || e == '/')
        {
            if (flag == 1)//如果在括号里面
            {
                if (Operator.top() == '(')
                {
                    Operator.push(e);
                }
                else if ((e == '*' || e == '/') && (Operator.top() ==
'+' || Operator.top() == '-'))
                {
                    Operator.push(e);
                }
                else//操作符的优先级低于或等于栈顶操作符则出栈，直至遇到'('
                {
                    while (Operator.top() != '(')
                    {
                        operand.push_back(Operator.top());
                        Operator.pop();
                    }
                    Operator.push(e);
                }
            }
            else if (Operator.empty())//栈空就入栈
            {
                Operator.push(e);
            }
            //操作符的优先级高于栈顶操作符，入栈
            else if ((e == '*' || e == '/') && (Operator.top() == '+' ||
Operator.top() == '-'))
            {
                Operator.push(e);
            }
            else//操作符的优先级低于或等于栈顶操作符则出栈，直至栈空或者优先级高于
栈顶操作符
            {
                while (!Operator.empty())
                {
                    operand.push_back(Operator.top());
                    Operator.pop();
                }
                Operator.push(e);
            }
        }
        //操作数
        else
        {
            operand.push_back(e);
        }
    }
    while (!Operator.empty())//最后将剩余在栈里面的全部出栈
    {
        operand.push_back(Operator.top());
        Operator.pop();
    }
```

```
    return operand;
}
```

- 将逆波兰表达式进行运算

```
int evalRPN(const string& s)
{
    stack<char> operand;
    int left = 0, right = 0;
    for (const auto& e : s)
    {
        if (e == '+' || e == '-' || e == '*' || e == '/')
        {
            switch (e)
            {
            case '+':
                right = operand.top();
                operand.pop();
                left = operand.top();
                operand.pop();
                operand.push(left + right);
                break;
            case '-':
                right = operand.top();
                operand.pop();
                left = operand.top();
                operand.pop();
                operand.push(left - right);
                break;
            case '*':
                right = operand.top();
                operand.pop();
                left = operand.top();
                operand.pop();
                operand.push(left * right);
                break;
            case '/':
                right = operand.top();
                operand.pop();
                left = operand.top();
                operand.pop();
                operand.push(left / right);
                break;
            }
        }
        else//操作数
        {
            operand.push(e - '0');
        }
    }
    return operand.top();
}
```

2. 用栈求解迷宫问题

4. 队列

> 一端进一端出（先进先出）

5.树

树中的节点数等于所有节点的度数之和加一

1. 二叉树

满二叉树：除叶子结点之外，所有分支都有双分节点，叶子都在最下面层

完全二叉树：最多只有下面两层的节点小于二，且最下面层的叶子结点从左往右依次排列

将树转化为二叉树：左边放孩子，右边放兄弟

森林转化为二叉树：增根节点，转化为二叉树，删除根节点

2. 顺序存储结构

将树补全为完全二叉树，用数组进行存储（如果不是完全二叉树，空间浪费严重）

3. 链式存储结构

二叉树：

```
struct node
{
    int data;
    node *rchild,*lchild;
}
```

4. 基本操作

1. 先序遍历
    - 迭代

    ```
    vector<int> preorderTraversal(TreeNode* root) {
            if(root==nullptr) return v;
            v.emplace_back(root->val); //输入数组语句
            preorderTraversal(root->left);
            preorderTraversal(root->right);
            return v;
        }
    ```

2. 中序遍历

    按照左、根、右的顺序进行遍历

```
//中序遍历函数
vector<int> inorderTraversal(TreeNode* root) {
    if(root==nullptr) return v;
    inorderTraversal(root->left);
    v.emplace_back(root->val);  //输入数组语句
    inorderTraversal(root->right);
    return v;
}
```

3. 后序遍历

4. 层次遍历

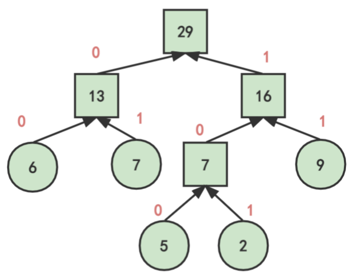
5. 线索二叉树

   充分利用空余存储空间，左空节点指向前驱，右空指针指向后继

6. 最优二叉树（赫夫曼树）

   - 树的路径长度：所有节点与根节点长度之和
   - 树的带权路径长度：所有节点与根节点的带权路径长度之和
   - 构造最优二叉树：
     1. 根据给定的 n 个权值 {w1 , w2 , ..., wn }，构造 n 棵二叉树的集合 F = {T1 , T2 , ... , Tn }，其中每棵二叉树中均只含一个带权 值为 wi 的根结点，其左、右子树为空树
     2. 在F 中选取其根结点的权值为最小和次小的两棵二叉树， 分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和
     3. 从F中删去这两棵树，同时加入刚生成的新树
     4. 重复 2 和 3 两步，直至 F 中只含一棵树为止
   - 赫夫曼编码

     将各字母出现的概率设为其权，进行排序，并构造成赫夫曼树



假设对A、B、C、D、E进行编码，已知他们的权重

字符	权重	编码
A	6	00
B	7	01
C	5	100
D	2	101
E	9	11

把要编码的字符权重作为叶子结点 ⇨ 构建赫夫曼树 ⇨ 左子树为0，右子树为1 从根节点到叶子结点累计编码

     从而可以得出赫夫曼编码：

   - 代码如下：

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
/*树结构*/
//用数组连续存储，不会浪费空间
//用数组下标存储左右孩子、父亲结点
typedef struct
{
```

```
    int weight;
    int left;
    int right;
    int parent;
}Node, * HuffmanTree;

typedef char* HuffmanCode;
void select(HuffmanTree* T, int n, int* m1, int* m2);

/*创建赫夫曼树*/
//传入n个权重，作为哈夫曼树的n个叶子结点
void CreateHuffmanTree(HuffmanTree* T, int w[], int n)
{
    int m = 2 * n - 1;//n个叶子结点，共m个结点
    int m1, m2;//用于建立下一个结点的两结点，值为最小的两个
    *T = (HuffmanTree)malloc((m + 1) * sizeof(Node));
    //初始化前n个结点（叶子结点），权重赋值，暂时没有左右孩子与父亲
    for (int i = 1; i <= n; i++)
    {
        (*T)[i].weight = w[i];
        (*T)[i].left = 0;
        (*T)[i].right = 0;
        (*T)[i].parent = 0;
    }
    //初始化[n+1,m]个结点(非叶子结点)
    for (int i = n + 1; i <= m; i++)
    {
        (*T)[i].weight = 0;
        (*T)[i].left = 0;
        (*T)[i].right = 0;
        (*T)[i].parent = 0;
    }
    //开始建树，第i个结点的两孩子为m1,m2，权重为两孩子结点权重之和
    for (int i = n + 1; i <= m; i++)
    {
        select(T, i - 1, &m1, &m2);
        (*T)[i].left = m1;
        (*T)[i].right = m2;
        (*T)[m1].parent = i;
        (*T)[m2].parent = i;
        (*T)[i].weight = (*T)[m1].weight + (*T)[m2].weight;

        printf("%d (%d %d)\n", (*T)[i].weight, (*T)[m1].weight, (*T)
[m2].weight);
    }
    printf("\n");
}

/*选取得到n个无父节点的两最小结点*/
void select(HuffmanTree* T, int n, int* m1, int* m2)
{
    int m;//存储最小值的数组下标
    //给m赋初值
    for (int i = 1; i <= n; i++)
    {
```

```
        if ((*T)[i].parent == 0)
        {
            m = i;
            break;
        }
    }
    //找到当前最小的权重（叶子结点）
    for (int i = 1; i <= n; i++)
    {
        if ((*T)[i].parent == 0 && (*T)[i].weight < (*T)[m].weight)
        {
            m = i;
        }
    }
    //先赋给m1保存一个，再去寻找第二小的值
    *m1 = m;
    for (int i = 1; i <= n; i++)
    {
        if ((*T)[i].parent == 0 && i != *m1)
        {
            m = i;
            break;
        }
    }
    for (int i = 1; i <= n; i++)
    {
        if ((*T)[i].parent == 0 && i != *m1 && (*T)[i].weight < (*T)
[m].weight)
        {
            m = i;
        }
    }
    //保存第二小的数
    *m2 = m;
}

/*创建哈夫曼编码*/
//从n个叶子结点到根节点逆向求解
void CreateHuffmanCode(HuffmanTree* T, HuffmanCode* C, int n)
{
    //编码长度为s-1,第s位为\0
    int s = n - 1;
    //当前结点的父节点数组下标
    int p = 0;
    //为哈夫曼编码分配空间(二维数组)
    C = (HuffmanCode*)malloc((n + 1) * sizeof(char*));
    //临时保存当前叶子结点的哈夫曼编码
    char* cd = (char*)malloc(n * sizeof(char));
    //最后一位为\0
    cd[n - 1] = '\0';

    for (int i = 1; i <= n; i++)
    {
        s = n - 1;
        //c指向当前结点,p指向此结点的父节点,两者交替上升，直到根节点
```

```
        for (int c = i, p = (*T)[i].parent; p != 0; c = p, p = (*T)
[p].parent)
        {
            //判断此结点为父节点的左孩子还是右孩子
            if ((*T)[p].left == c)
                cd[--s] = '0';//左孩子就是编码0
            else
                cd[--s] = '1';//右孩子就是编码1
        }
        //为第i个编码分配空间
        C[i] = (char*)malloc((n - s) * sizeof(char));
        //将此编码赋值到整体编码中
        strcpy(C[i], &cd[s]);
    }
    //释放
    free(cd);
    //打印编码序列
    for (int i = 1; i <= n; i++)
    {
        printf("%d %s", (*T)[i].weight, C[i]);
        printf("\n");
    }
}

int main()
{
    HuffmanTree T;
    HuffmanCode C;
    int n, w1, * w;
    scanf_s("%d", &n);
    w = (int*)malloc((n + 1) * sizeof(int));
    for (int i = 1; i <= n; i++)
    {
        scanf_s("%d", &w1);
        w[i] = w1;
    }
    printf("\n");
    CreateHuffmanTree(&T, w, n);
    CreateHuffmanCode(&T, &C, n);
    return 0;
}
```

6. 图

1. 基本概念

详细内容参考《离散数学》

完全图：n(n-1)/2 条边的无向图；（C(n，2))

链、简单路径、回路、简单回路（回路&&简单路径）

连通图、强连通图（有向图）

- 生成树、生成森林：

生成树：一个连通图G的一个包含**所有顶点**的**极小连通子图**T（T是一个有n顶点，n-1条边的生成树）

生成森林：类似于生成树

2. 存储结构

1. 顺序存储

邻接矩阵（有用1，没有用0）

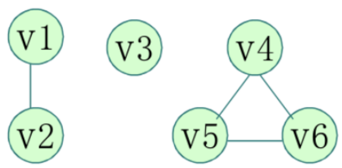
- 对于无向图来说，这是一个沿对角线的对称矩阵
- 操作很方便，但是内存浪费
- 当然，对于带权网来说，不以1,0为存储内容，而是以权值和0（用无穷大更贴切）作为存储内容

```
#define MaxVertexNum 100                    //顶点数目最大值
typedef struct{
    char Vex[MaxVertexNum];                 //顶点表
    int Edge[MaxVertexNum][MaxVertexNum];    //邻接矩阵边表
    int vexnum.arcnum;                     //图的当前顶点数和边数/弧数
}MGraph;
```

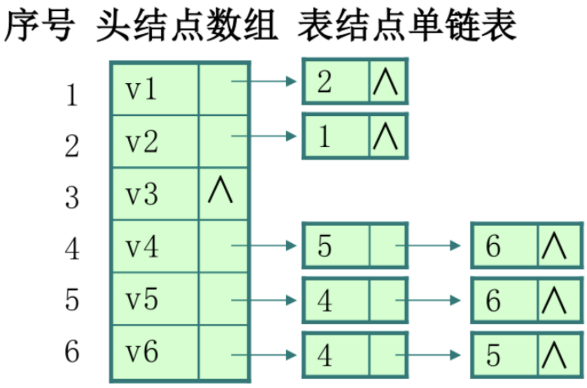
2. 链式存储

邻接表、邻接多重表、十字链表

- 邻接表：分为头结点和表节点



图G



图G邻接表

有向图原理相同，都是由一个头结点和多个表节点组成

为了方便求顶点的入度，故引出逆邻接表

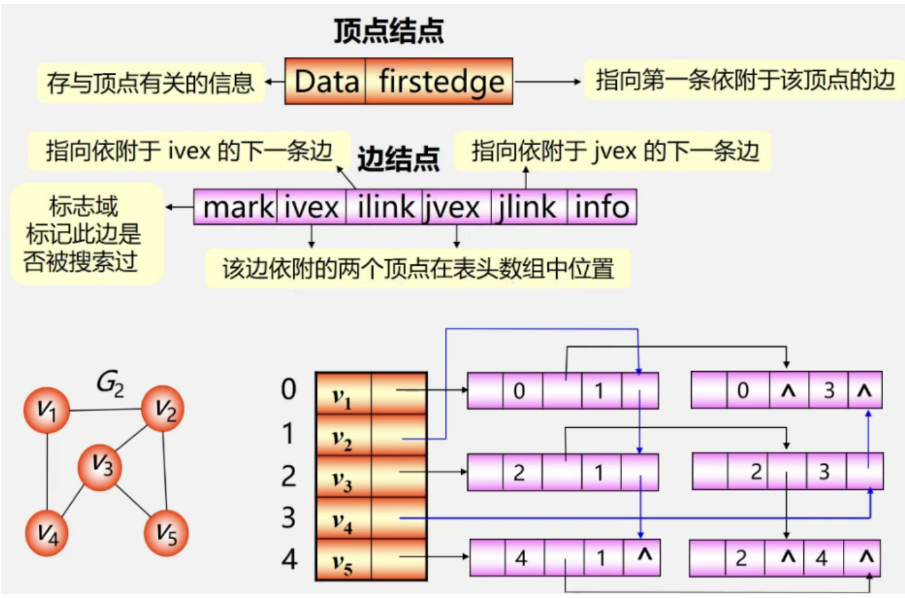
```
#include <iostream>
#include <stdio.h>
#define MAXVEX 1000          //最大顶点数
typedef char VertexType;         //顶点类型
typedef int EdgeType;            //边上权值类型

typedef struct EdgeNode          //边表结点
{
    int adjvex;          //邻接点域，存储该顶点对应的下标
    EdgeType weigth;         //用于存储权值，对于非网图可以不需要
    struct EdgeNode *next;       //链域，指向下一个邻接点
}EdgeNode;
```

```
typedef struct VertexNode       //顶点表结构
{
    VertexType data;        //顶点域，存储顶点信息
    EdgeNode *firstedge;        //边表头指针
}VertexNode, AdjList[MAXVEX];

typedef struct
{
    AdjList adjList;
    int numVertexes, numEdges;  //图中当前顶点数和边数
}GraphList;
//建立图的邻接表结构
void CreateGraph(GraphList *g)
{
    int i, j, k;
    EdgeNode *e;
    printf("输入顶点数和边数:\n");
    scanf("%d%d", &g->numVertexes, &g->numEdges);
    getchar();
    for(i = 0; i <g->numVertexes; i++)
    {
        printf("请一次一个输入顶点%d:\n", i);
        scanf("%c",&g->adjList[i].data);          //输入顶点信息
        getchar();
        g->adjList[i].firstedge = NULL;          //将边表置为空表
    }
    g->adjList[i].firstedge = NULL;
    //建立边表
    for(k = 0; k < g->numEdges; k++)//关于邻接表的循环次数无向图与与有向图都是g->numEdges次
    {
        printf("输入无向图边(vi,vj)上的顶点序号和权值:\n");
        int w;
        scanf("%d%d%d",&i,&j,&w);
        e =new EdgeNode;
        e->adjvex = j;        //邻接序号为j
        e->weigth = w;        //边<vi,vj>的权值
        e->next = g->adjList[i].firstedge;//将e指针指向当前顶点指向的结构
        g->adjList[i].firstedge = e;//将当前顶点的指针指向e
        //这里两次类似的代码就体现了邻接表的重复的弊端
        e = new EdgeNode;
        e->adjvex =i;
        e->weigth = w;        //边<vj,vi>的权值
        e->next = g->adjList[j].firstedge;
        g->adjList[j].firstedge = e;
    }
}
void printGraph(GraphList *g)
{
    int i = 0;

    while(g->adjList[i].firstedge != NULL && i < MAXVEX)
    {
        printf("顶点:%c\n", g->adjList[i].data);
```

```
        EdgeNode *e = NULL;
        e = g->adjList[i].firstedge;
        while(e != NULL)
        {
            if(e->adjvex!=i)
            printf("邻接点下标:%d  边:<%c,%c>  weigth: %d\n", e->adjvex,g->adjList[i].data,g->adjList[e->adjvex].data,e->weigth);
            e = e->next;
        }
        i++;
        printf("\n");
    }
}
int main(int argc, char **argv)
{
    GraphList g;
    CreateGraph(&g);
    printGraph(&g);
    return 0;
}
```

- 邻接多重表

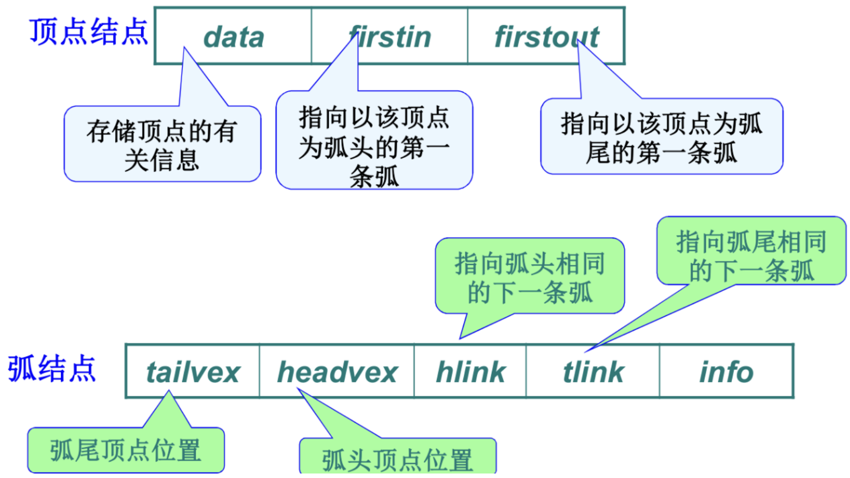
  用于存储无向图，相对于邻接表（n+2e）可以节约内存

  有点晕 😴 

- 十字链表

  将邻接表和逆邻接表结合在一起即组成了十字链表（比较复杂😭 ）

十字链表的基本结构：



3. 深度优先

类似于树的先序遍历

1.创建一个visited数组，用于记录所有被访问过的顶点。

2.从图中v0出发，访问v0。

3.找出v0的第一个未被访问的邻接点，访问该顶点。以该顶点为新顶点，重复此步骤，直至刚访问过的顶点没有未被访问的邻接点为止。

4.返回前一个访问过的仍有未被访问邻接点的顶点，继续访问该顶点的下一个未被访问领接点。

5.重复2,3步骤，直至所有顶点均被访问，搜索结束。

因此，这是一个**递归**的算法！

```
//这个递归太巧秒了！！！
//以邻接表为基础
void DFS(GraphAdjlist *G,int i)
{    EdgeNode *p;
     visited[i]=1;
     printf("%c ",G->adjlist[i].data);
     p=G->adjlist[i].firstedge;
     while(p!=NULL)
     {    if(visited[p->adjvex]==0)
            DFS(G,p->adjvex);
          p=p->next;//当此路上所有的节点都被找完之后，执行此行代码，寻找原来的没找的分路
     }
}
void DFSTraverse(GraphAdjlist *G)
{    int i;
     for(i=0;i<G->n;i++)
       visited[i]=0;
     for(i=0;i<G->n;i++)
       if(visited[i]==0)
         DFS(G,i);
}
```

4. 广度优先

先遍历一个节点，然后遍历那个节点所连接的的周边节点，之后再一个结点一个结点的往外遍历，重复循环

代码略（思路简单，有点难写）

5. Prim算法

图生成最小生成树（更适合稠密网）时间复杂度与节点相关（n^2）

步骤：

1.选择一个起始顶点作为已加入最小生成树的节点集合。

2.将其余顶点作为未加入最小生成树的节点集合。

3.重复以下步骤直至所有顶点都被加入最小生成树：

在已加入最小生成树的节点集合中的节点中找到与之相连的、但不在该集合中的距离最小的节点（即最短边）。将这个节点加入最小生成树，并更新该节点相关的边的权值。

4.终止条件：当所有顶点都被加入最小生成树后，算法结束，得到最小生成树。

6. Kruskal算法

更适合稀疏网（时间复杂度与边相关（e * log e））

1.初始化：将所有边按照权值从小到大进行排序。

2.并查集初始化：将图中的每个顶点都视为一个独立的集合。

3.遍历排序后的边：对于排序后的每一条边，执行以下操作：

判断这条边的两个顶点是否属于同一个集合（即是否已经连通）。

a.如果两个顶点属于不同的集合（即未连通），则将这条边加入最小生成树中，并将两个集合合并。

b.如果两个顶点属于同一个集合（即已连通），则跳过这条边，继续下一条边的判断。

4.结束条件：当已经选择了n-1条边（n为顶点数）时，算法结束，此时得到的就是最小生成树。

7. 带权图的最短路径

1. 单个节点到其他节点（Dijkstra算法）

贪心算法：局部最优计算全局最优

2. 每一对节点

重复执行n次Dijkstra算法

二、数论

1. 因式分解与算数基本定理

p是素数，假设P|$a_1a_2a_3$......，则p至少整除其中一个数

每个整数（n>=2）可唯一分解成素数的乘积（n=$P_1P_2P_3$......P_r）

2. 同余式

$ax \equiv b \pmod n$ 的方程称为 线性同余方程

如：求关于 x 的同余方程 ax≡1(modb)的最小正整数解（这里同时也用到了extend_gcd拓展欧几里得定理）。

```
#include<iostream>
using namespace std;
typedef long long ll;
int extend_gcd(ll a,ll b,ll &x,ll &y)
{
    if(b==0)
    {
        x=1;
        y=0;
        return a;
    }
    else
    {
        ll g=extend_gcd(b,a%b,x,y);
        ll t=x;
        x=y;
        y=t-(a/b)*y;
        return g;
    }
}
int main()
{
    ll x,y;
    ll a,b;
    cin>>a>>b;
    ll g = extend_gcd(a,b,x,y);
    cout<<(x%b + b) % b;
}
```

详解：[AcWing203 同余方程-CSDN博客]

3. 欧拉函数

> 定义：对于一个正整数n，n的欧拉函数φ (n)，表示小于等于n与n互质的正整数的个数

1. 基于素因式分解求欧拉函数的算法

假设 p_1，p_2，⋯，p_r 是整除 m 的不同素数．证明 $\phi(m)$ 的下述公式成立：

$$\phi(m) = m\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right).$$

```
int sigma(int n)
{
    int ans = n;
    for (int i = 2; i * i <= n; i++)
    {
        if (n % i == 0)
        {
            ans = ans / i * (i - 1);
```

```
            while (n % i == 0) n /= i;
        }
    }
    //最后可能会剩下一个没有被处理的素数
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}
```

当然，也可以先把素数筛出来再进行处理

4.幂取模（快速幂）

形如：$a^n \bmod m$

在求解时要用到快速幂

$$pow(x,n)=\begin{cases}1 & (n=0\text{时})\\ pow\left(x^2,\frac{n}{2}\right) & (n\text{为偶数时})\\ pow\left(x^2,\frac{n}{2}\right)x & \left(n\text{为奇数时}\right)\end{cases}$$

上述算法称之为“快速幂”，时间复杂度优化到 $O(\log n)$ 。

```
long long quick_power(long long a, long long b, long long mod)
{
    if (a == 1 || b == 0) return 1;
    else return b%2==1 ? a*(quick_power(a*a%mod, b/2, mod))%mod :
quick_power(a*a%mod, b/2, mod);
}
```

三、算法

1. 动态规划