快速排序算法模板 —— 模板题 AcWing 785. 快速排序

```
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)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; 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 —— 模板题 AcWing 787. 归并排序

```
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 —— 模板题 AcWing 789. 数的范围

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

// 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 —— 模板题 AcWing 790. 数的三次方根

```
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 —— 模板题 AcWing 791. 高精度加法

```
// 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 —— 模板题 AcWing 792. 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;
}
```

高精度乘低精度 —— 模板题 AcWing 793. 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;
}
```

高精度除以低精度 —— 模板题 AcWing 794. 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 —— 模板题 AcWing 795. 前缀和

```
S[i] = a[1] + a[2] + ... a[i]
a[l] + ... + a[r] = S[r] - S[l - 1]
```

二维前缀和 —— 模板题 AcWing 796. 子矩阵的和

```
S[i, j] = 第i行j列格子左上部分所有元素的和
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]
```

一维差分 —— 模板题 AcWing 797. 差分

```
给区间[l, r]中的每个数加上c: B[l] += c, B[r + 1] -= c
```

二维差分 —— 模板题 AcWing 798. 差分矩阵

```
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:
S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
```

位运算 —— 模板题 AcWing 801. 二进制中1的个数

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

双指针算法 —— 模板题 AcWIng 799. 最长连续不重复子序列, AcWing 800. 数组元素的目标和

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

    // 具体问题的逻辑
}
常见问题分类:
    (1) 对于一个序列，用两个指针维护一段区间
    (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作
```

离散化 —— 模板题 AcWing 802. 区间和

```
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
}
```

区间合并 —— 模板题 AcWing 803. 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)
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);

    if (st != -2e9) res.push_back({st, ed});

    segs = res;
}
```

数据结构

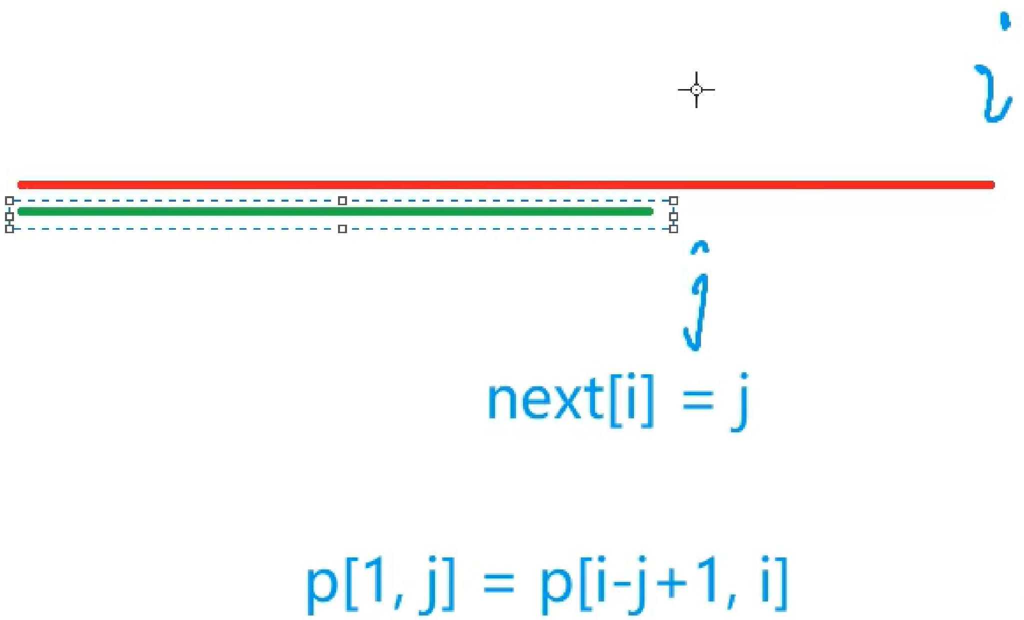
双链表

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <string>

using namespace std;

const int N = 100010;
int e[N], l[N], r[N], idx;

void insert(int k, int x)
{
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;
    r[k] = idx ++ ;
}

void remove(int k)
{
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}

int main()
{
    r[0] = 1, l[1] = 0;
    idx = 2;

    int T;
```

```
    cin >> T;

    while (T -- )
    {
        string op;
        int x;
        cin >> op >> x;
        if (op == "L") insert(0, x);
        else if (op == "R") insert(l[1], x);
        else if (op == "D") remove(x + 1);
        else if (op == "IL")
        {
            int t;
            cin >> t;
            insert(l[x + 1], t);
        }
        else if (op == "IR")
        {
            int t;
            cin >> t;
            insert(x + 1, t);
        }
    }

    for (int i = r[0]; i != 1; i = r[i])
    {
        printf("%d ", e[i]);
    }

    return 0;
}
```

KMP字符串

- `next[i] = j` 的含义是模式字符串中 `p[1, j]` 和 `p[i - j + 1, i]` 是相等的。也就是 **第一个字符到第j的字符** 组成的字符串和 **第i - j + 1到第i的字符是相等的** 。



- `next` 在某个头文件里面有，所以起名为 `ne` 才能万无一失。

- 代码如下

```cpp
#include <iostream>

using namespace std;

const int N = 100010, M = 1000010;

char p[N], s[M];
int n, m;
int ne[N];

int main()
{
    cin >> n >> p + 1 >> m >> s + 1;

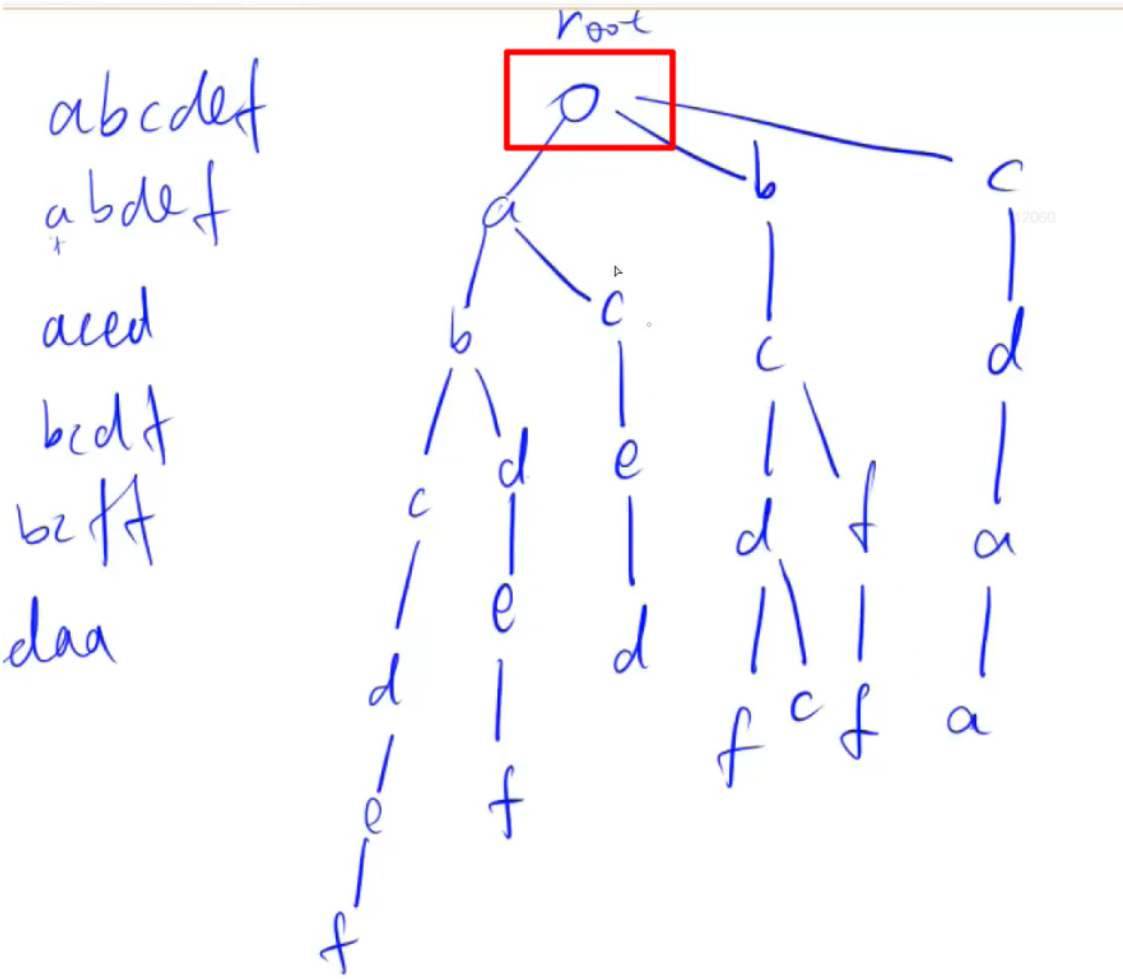
    for (int i = 2, j = 0; i <= n; i ++ )
    {
        while (j && p[j + 1] != p[i]) j = ne[j];
        if (p[j + 1] == p[i]) j ++ ;
        ne[i] = j;
    }

    for (int i = 1, j = 0; i <= m; i ++ )
    {
        while (j && p[j + 1] != s[i]) j = ne[j];
        if (p[j + 1] == s[i]) j ++ ;
        if (j == n)
        {
            // 匹配成功
            printf("%d ", i - n); // 减去模式字符串的长度等于起点
            j = ne[j];
        }
    }

    return 0;
}
```

TRIE树

- `trie` 树的根节点是一个空结点，所以存储的时候要使用 `++ idx` 而不是 `idx ++`



- 因为一个字符串的末尾是 `\0` ，所以可以这样遍历字符串

```
for (int i = 0; str[i]; i ++ )
```

- 代码如下

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

const int N = 20010;

char str[100010];

// son[p][u]表示u这个点下一个字母是u + 'a'的子节点的下标
int son[N][26 + 1], cnt[N], idx; // 下标是0的点既是根节点又是空结点

void insert()
{
    // p用来存下一个探索的节点的下标。
    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] ++ ;
}

int query()
{
    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];
}

int main()
{
    int T;
    cin >> T;
    while (T -- )
    {
        char op[2];
        cin >> op >> str;
        if (*op == 'I') insert();
        else cout << query() << endl;
    }

    return 0;
}
```

最大异或对

- 异或相当于不进位加法
- ```
  for (int i = 30; i >= 0; i -- )
  ```

  可以相当于

  ```
  for (int i = 30; ~i; i -- )
  ```
- 代码如下

  ```
  #include <iostream>
  #include <stdio.h>
  #include <limits.h>
  #include <string.h>
  #include <algorithm>

  using namespace std;

  const int M = 3100010, N = 100010;

  int a[N], son[M][2], idx, n;

  void insert(int x)
  ```
  ```

```
{
    int p = 0;
    for (int i = 30; ~i; i -- )
    {
        int &s = son[p][x >> i & 1];
        // 这里一定要是++idx，不然第一个下标不会被用到
        if (!s) s = ++ idx;
        p = s;
    }
}

int query(int x)
{
    int res = 0;
    int p = 0;
    for (int i = 30; ~i; i -- )
    {
        int s = x >> i & 1;
        // 如果有能让这一位为1的路可走
        if (son[p][!s])
        {
            // 异或为1，加上相应的数字
            res += 1 << i;
            p = son[p][!s];
        }
        // 如果没有则只能走另一条路了
        else p = son[p][s];
    }

    return res;
}

int main()
{
    cin >> n;
    for (int i = 0; i < n; i ++ )
    {
        cin >> a[i];
        insert(a[i]);
    }
    int res = INT_MIN;
    for (int i = 0; i < n; i ++ ) res = max(res, query(a[i]));

    cout << res << endl;

    return 0;
}
```

堆

- 如果需要删除一个元素，要将队尾的元素移到这个位置然后通过 `down` 进行调整。因为在堆中间删除元素非常不便，但是删除最后一个元素非常容易。所以将最后一个元素移动到这个位置再 `down` 进行调整以达到删除元素的效果。
- 这里要注意下标应该从1开始，因为 `0 >> 1 == 0` 容易出问题。

- 不要用 `size` 来做变量名，会CE。
- 堆的节点是按顺序一层一层往下排的，一个节点的左右子节点是 `n * 2` 和 `n * 2 + 1`。

BruceChen 2个月前 回复
可以用 `scanf(" %c", &op);`
加入' '会匹配零个或多个 空格、制表符、换行符

- Eurus 1个月前 回复
  真的诶，我学了这么久才发现！！！！o(╥﹏╥)o

  yxc 1个月前 回复
  👍

  samwu 22天前 回复
  厉害

如何手写一个堆？

1. 插入一个数	heap[++ size] = x; up(size);
2. 求集合当中的最小值	heap[1];
3. 删除最小值	heap[1] = heap[size]; szie -- ; down(1);
4. 删除任意一个元素	heap[k] = heap[size]; size -- ; down(k); up(k);
5. 修改任意一个元素	heap[k] = x; down(k); up(k);

堆排序

- 不用写 `up` 操作。
- 代码如下

```
#include <iostream>
#include <stdio.h>
#include <algorithm>

using namespace std;

const int N = 100010;

int cnt, n, m, h[N];

void down(int u)
{
    // 用来记录这个点和他的左右节点中哪一个点最小
    int t = u;

    // 如果u * 2小于堆中元素数量而且小于h[t]，则将t更新
    if (u * 2 <= cnt && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] < h[t]) t = u * 2 + 1;

    if (u != t)
    {
```

```
        swap(h[u], h[t]);
        down(t);
    }
}

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i ++ ) cin >> h[i];
    cnt = n;

    // 从倒数第二层开始down, 一直到最顶层，时间复杂度O(n)
    for (int i = n >> 1; i; i -- ) down(i);

    while (m -- )
    {
        printf("%d ", h[1]);
        h[1] = h[cnt -- ];
        down(1);
    }

    return 0;
}
```

模拟堆

- 这道题有坑，要先存下 `k = ph[k]` 再进行后续操作，因为 `ph[k]` 会改变，而我们需要 `modify` 的元素的位置不变
- 代码如下

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

const int N = 100010;

// ph用来代表指针(pointer)到heap的映射，hp用来代表heap到指针的映射
// ph[i] = j就相当于第i个插入的数在h中的下标是j
int h[N], ph[N], hp[N], idx, cnt, m;

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 <= cnt && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (t != u)
```

```
    {
        heap_swap(u, t);
        down(t);
    }
}

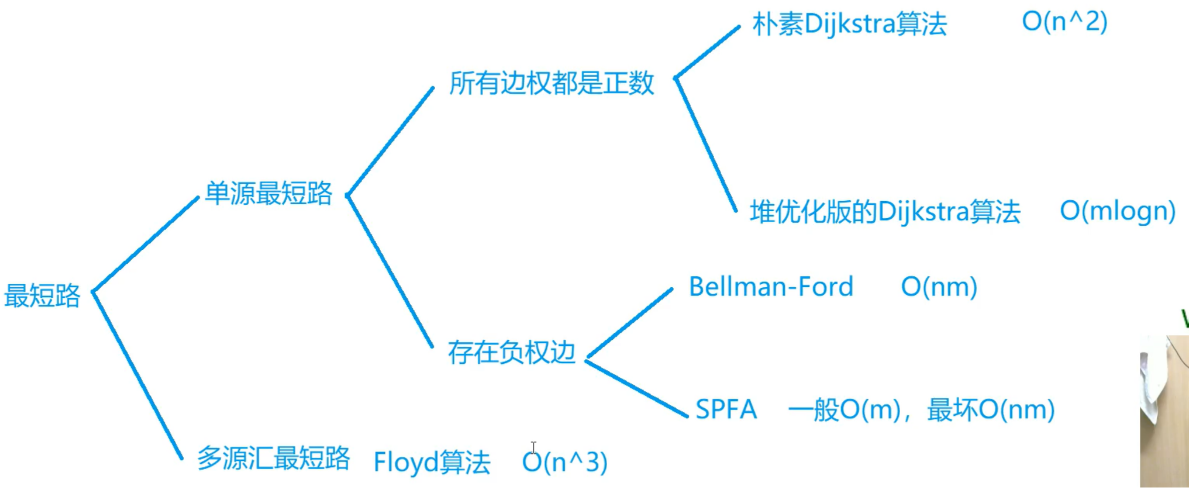
void up(int u)
{
    while (u / 2 && h[u / 2] > h[u])
    {
        heap_swap(u / 2, u);
        u /= 2;
    }
}

int main()
{
    int T;
    cin >> T;
    while (T -- )
    {
        string op;
        cin >> op;
        if (op == "I")
        {
            int x;
            cin >> x;
            cnt ++ , m ++ ;
            ph[m] = cnt, hp[cnt] = m;
            h[cnt] = x;
            up(cnt);
        }
        else if (op == "PM")
            cout << h[1] << endl;
        else if (op == "DM")
        {
            heap_swap(1, cnt);
            cnt -- ;
            down(1);
        }
        else if (op == "D")
        {
            int k;
            cin >> k;/*
            heap_swap(cnt, ph[k]);
            cnt -- ;
            down(ph[k]), up(ph[k]);*/
            // 这里上面注释掉的代码AC不了，因为在heap_swap的时候ph[k]改变了，之后down
和up操作的对象不一样了
            // printf("ph[k] = %d\n", ph[k]);
            k = ph[k];
            heap_swap(cnt, k);
            // printf("ph[k] = %d\n", ph[k]);
            cnt -- ;
            down(k), up(k);
        }
        else
        {
```

```
            int k, x;
            cin >> k >> x;
            h[ph[k]] = x;
            down(ph[k]), up(ph[k]);
            //heap_swap(1, ph[k]);
            //h[1] = x;
            //down(1);
        }
    }

    return 0;
}
```

搜索与图论



（一）

八皇后

> **输入格式**
>
> 共一行，包含整数n。
>
> **输出格式**
>
> 每个解决方案占n行，每行输出一个长度为n的字符串，用来表示完整的棋盘状态。
>
> 其中"."表示某一个位置的方格状态为空，"Q"表示某一个位置的方格上摆着皇后。
>
> 每个方案输出完成后，输出一个空行。
>
> 输出方案的顺序任意，只要不重复且没有遗漏即可。
>
> **数据范围**
>
> $1 \le n \le 9$
>
> **输入样例:**
>
> ```
> 4
> ```
>
> **输出样例:**
>
> ```
> .Q..
> ...Q
> Q...
> ..Q.
>
> ..Q.
> Q...
> ...Q
> .Q..
> ```

- 这道题先按照最原始的思路来想吧，第二种方法的斜对角线和反斜对角线的判断有点不太理解。
- 判断是否在同一对角或者反对角线的方法就是 **看截距** 。比如经过点 `(1, 1)` 的对角线和反对角线的截距都是 `1 + 1 == 2` ，则 `dg[x + y] = udg[x + y] = true` 即可。即 `dg[2] = udg[2] = true` 。这里dg是 *diagonal* （对角线）的意思。
- 当然这里 `(0, 9)` 和 `(9, 0)` 会标记在同一个斜对角线和反斜对角线上，但是事实上他们并不在同一个斜对角线而只在同一个反斜对角线上。不过既然他们在同一反斜对角线上，这种条件必然不可以选，所以这样写也是没问题的。

  ```
  #include <iostream>
  #include <stdio.h>
  #include <string.h>
  #include <stdlib.h>

  using namespace std;

  const int N = 20;
  bool col[N], row[N], dg[N * N], udg[N * N];
  int n;
  char g[N][N];

  void dfs(int x, int y, int s)
  {
      if (s > n) return;
      if (y == n) x ++ , y = 0;
  ```

```
    if (x == n)
    {
        if (s == n)
        {
            for (int i = 0; i < n; i ++ ) puts(g[i]);
            puts("");
        }
        return;
    }

    g[x][y] = '.';
    dfs(x, y + 1, s); // 不要这一个点

    if (!col[y] && !row[x] && !dg[x + y] && !udg[x - y + n]) // 如果可以要这个点
    {
        col[y] = row[x] = dg[x + y] = udg[x - y + n] = true;
        g[x][y] = 'Q';
        dfs(x, y + 1, s + 1);
        g[x][y] = '.';
        col[y] = row[x] = dg[x + y] = udg[x - y + n] = false;
    }

}

int main()
{
    cin >> n;
    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < n; j ++ )
            g[i][j] = '.';
    dfs(0, 0, 0);
    return 0;
}
```

图

- 在图论题里面计算时间复杂度的时候一般把 **点的个数** 记作 `n` ，把 **边的条数** 记作 `m` 。
- 树是一种特殊的图（无环联通图），无向图也是一种特殊的有向图。
- Y总的邻接表是用 **链表** 来实现的，和算法笔记里面用 `vector<int>` 来存稍有不同。（Y总说是因为 `vector` 的速度比数组模拟要慢一些。）
- 链表添加的模板

  ```
  void add(int a, int b)
  {
      e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
  }
  ```

- 树和图深度优先搜索的模板

```cpp
int h[N], e[M], ne[M], idx;
bool st[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}


void dfs(int u)
{
    st[u] = true; // 标记一下，已经被搜过了

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}


int main()
{
    memset(h, -1, sizeof h);


    dfs(1);
}
```

- 树和图的DFS和BFS的时间复杂度都跟 **点数和边数** 成线性关系 `O(n + m)` 。因为每个点都只会被遍历到一次。

树的重心

给定一颗树，树中包含n个结点（编号1~n）和n-1条无向边。

请你找到树的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。

重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。

输入格式

第一行包含整数n，表示树的结点数。

接下来n-1行，每行包含两个整数a和b，表示点a和点b之间存在一条边。

输出格式

输出一个整数m，表示将重心删除后，剩余各个连通块中点数的最大值。

数据范围

$1 \le n \le 10^5$

输入样例

```
9
1 2
1 7
1 4
2 8
2 5
4 3
3 9
4 6
```

输出样例：

```
4
```

- 这道题 `int dfs(int u)` 返回的是这个从这个节点往下走有多少个节点

```
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

const int N = 100010, M = N * 2; // 边的条数可能比节点个数多
int h[N], e[M], ne[M], n, idx;
int ans = N;
bool st[M];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx;
    idx ++ ;
}

// 返回的是u这个点往下DFS有多少个节点
int dfs(int u)
{
    st[u] = true;

    int sum = 1, res = 0; // sum用来返回dfs的结果（这个点往下有多少个点(包括这个点本身)），res用来存储这个点子块的最大值
```

```cpp
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (st[j]) continue;

        int s = dfs(j); // 从这个点往下有多少个点
        res = max(res, s);
        sum += s; // 累加结果
    }
    res = max(res, n - sum); // 上面的子块
    ans = min(res, ans);
    return sum;
}

int main()
{
    cin >> n;
    memset(h, -1, sizeof h);
    for (int i = 0; i < n; i ++ )
    {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }
    dfs(1);
    cout << ans << endl;
    return 0;
}
```

图中点的层次

给定一个n个点m条边的有向图，图中可能存在重边和自环。

所有边的长度都是1，点的编号为1~n。

请你求出1号点到n号点的最短距离，如果从1号点无法走到n号点，输出-1。

输入格式

第一行包含两个整数n和m。

接下来m行，每行包含两个整数a和b，表示存在一条从a走到b的长度为1的边。

输出格式

输出一个整数，表示1号点到n号点的最短距离。

数据范围

$1 \le n, m \le 10^5$

输入样例：

```
4 5
1 2
2 3
3 4
1 3
1 4
```

输出样例：

```
1
```

- 图的bfs，和普通bfs没有很大区别

```
#include <iostream>
#include <stdio.h>
#include <algorithm>
#include <string.h>
#include <queue>

using namespace std;

const int N = 100010, M = N * 2;
int e[N], ne[N], h[N], idx;
int d[N];
int n, m;

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}
```

```
int bfs()
{
    memset(d, -1, sizeof d);
    queue<int> q;
    q.push(1);
    d[1] = 0;
    while (q.size())
    {
        int t = q.front();
        q.pop();
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (d[j] == -1)
            {
                d[j] = d[t] + 1;
                q.push(j);
            }
        }
    }
    return d[n];
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for (int i = 0; i < m; i ++ )
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
    }
    cout << bfs();
    return 0;
}
```

有向图的拓扑序列

给定一个n个点m条边的有向图，点的编号是1到n，图中可能存在重边和自环。

请输出任意一个该有向图的拓扑序列，如果拓扑序列不存在，则输出-1。

若一个由图中所有点构成的序列A满足：对于图中的每条边(x, y)，x在A中都出现在y之前，则称A是该图的一个拓扑序列。

输入格式

第一行包含两个整数n和m

接下来m行，每行包含两个整数x和y，表示存在一条从点x到点y的有向边(x, y)。

输出格式

共一行，如果存在拓扑序列，则输出任意一个合法的拓扑序列即可。

否则输出-1。

数据范围

$1 \le n, m \le 10^5$

输入样例：

```
3 3
1 2
2 3
1 3
```

输出样例：

```
1 2 3
```

- 只有有向图有拓扑序列，无向图没有拓扑序列。
- 如果有环则不能构成拓扑序列。（因为没有顺序）
- 可以证明，一个有向无环图一定存在拓扑序列。所以有向无环图也被称为拓扑图。
- 有多少条边指向自己就是一个点的入度，有多少条边出去就是一个点的出度。
- 所有入度为0的点都可以作为起点。（因为不会有任何一个点会在前面）
- 一个有向无环图**至少存在一个入度为0的点。**
- 这道题首先遍历所有入度为0的点（这里有点问题，如果输入是这样的那输出是正确的吗）

```
4 2
1 2
3 4
```

输出

```
1 3 2 4
```

并将其入队，然后进行BFS，每BFS到一个点就删除这条边。这样如果BFS结束后所有的点都进入过队列，那么说明这组数据可以得到拓扑序列，合法。

```
#include <iostream>
#include <stdio.h>
#include <algorithm>
#include <string.h>
```

```
using namespace std;

const int N = 100010, M = N * 2;
int n, m;
int e[N], ne[N], h[N], idx;
int q[N];
int d[N]; // 记录每个点的入度

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

bool topsort()
{
    int hh = 0, tt = -1;
    for (int i = 1; i <= n; i ++ )
        if (d[i] == 0) // 遍历所有
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            d[j] -- ;
            if (d[j] == 0)
                q[ ++ tt] = j;
        }
    }
    return (tt == n - 1); // 如果所有元素都入过队，那么说明无环。
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for (int i = 0; i < m; i ++ )
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        d[b] ++ ;
    }

    if (topsort())
    {
        for (int i = 0; i < n; i ++ ) printf("%d ", q[i]);
        cout << endl;
    }
    else
        puts("-1");

    return 0;
}
```

（二）

- 最短路的难度在于建图，将题目抽象出图。

dijkstra

给定一个n个点m条边的有向图，图中可能存在重边和自环，所有边权均为正值。

请你求出1号点到n号点的最短距离，如果无法从1号点走到n号点，则输出-1。

输入格式

第一行包含整数n和m。

接下来m行每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。

输出格式

输出一个整数，表示1号点到n号点的最短距离。

如果路径不存在，则输出-1。

数据范围

$1 \le n \le 500$,
$1 \le m \le 10^5$,
图中涉及边长均不超过10000。

输入样例：

```
3 3
1 2 2
2 3 1
1 3 4
```

输出样例：

```
3
```

- `dijkstra`**一定不能存在负权边**
- 在图论题里面计算时间复杂度的时候一般把 **点的个数** 记作 `n` ，把 **边的条数** 记作 `m` 。
- 朴素版 `dijkstra` 算法的时间复杂度是 `O(n ^ 2)` ，堆优化版的 `dijkstra` 算法的时间复杂度是 `O(mlogn)` 。所以对于稠密图一般使用朴素版 `dijkstra` ，对于稀疏图一般使用堆优化版 `dijkstra` 。
-

朴素 Dijkstra　　　　S：当前已确定最短距离的点

① dist[1] = 0，dist[i] = +∞

② for i：1 ~ n

t ← 不在s中的，距离最近的点

s ← t

用t更新其它点的距离

- 朴素 `dijkstra` 这道题是一个稠密图，所以可以用邻接矩阵来做。

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

const int N = 510;
int g[N][N], dist[N];
bool st[N];
int n, m;

int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0; // 1到1的距离为0

    for (int i = 0; i < n - 1; i ++ ) // 迭代n次
    {
        int t = -1; // t是当前能到达的最近的点
        // 遍历所有的点，找出当前能到达的最近的点
        // 这里有两重遍历，是朴素版dijkstra算法中最慢的部分。主要是为了找到离当前点距离最短的点。
        for (int j = 1; j <= n; j ++ )
        {
            // 如果这个点没有走过或者这个点比之前的距离要短，则更新为这个点
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;
        }
        for (int j = 1; j <= n; j ++ )
        {
            // 如果从t这个点到这个点比较近，那么更新其最短距离
            dist[j] = min(dist[j], dist[t] + g[t][j]);
        }

        // 把t标记为已访问
        st[t] = true;
    }

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

```cpp
int main()
{
    memset(g, 0x3f, sizeof g);
    cin >> n >> m;
    while (m -- )
    {
        int x, y, z;
        cin >> x >> y >> z;
        g[x][y] = min(g[x][y], z); // 保留最短的边
    }

    cout << dijkstra();
    return 0;
}
```

- 堆优化版的 `dijkstra`

  https://www.acwing.com/solution/content/6291/

  朴素 `dijkstra` 最慢的地方**就在找到当前能走到的距离最近的点(O(n ^ 2))**。而堆优化版的 `dijkstra` 用堆优化了这里。

  > 给定一个n个点m条边的有向图，图中可能存在重边和自环，所有边权均为非负值。
  >
  > 请你求出1号点到n号点的最短距离，如果无法从1号点走到n号点，则输出-1。
  >
  > **输入格式**
  >
  > 第一行包含整数n和m。
  >
  > 接下来m行每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。
  >
  > **输出格式**
  >
  > 输出一个整数，表示1号点到n号点的最短距离。
  >
  > 如果路径不存在，则输出-1。
  >
  > **数据范围**
  >
  > $1 \le n, m \le 1.5 \times 10^5$,
  > 图中涉及边长均不小于0，且不超过10000。
  >
  > **输入样例：**
  >
  > ```
  > 3 3
  > 1 2 2
  > 2 3 1
  > 1 3 4
  > ```
  >
  > **输出样例：**
  >
  > ```
  > 3
  > ```

- 朴素版 `dijkstra` 算法的时间复杂度是 `O(n ^ 2)` ，堆优化版的 `dijkstra` 算法的时间复杂度是 `O(mlogn)` 。

```cpp
#include <iostream>
```

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
#include <map>
#include <queue>

using namespace std;
typedef pair<int, int> pii;
#define xx first
#define yy second

const int N = 1000010;

bool st[N];
int e[N], ne[N], w[N], h[N], idx;
int n, m;
int dist[N];

void add(int a, int b, int c)
{
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx ++ ;
}

int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    priority_queue<pii, vector<pii>, greater<pii>> heap;

    // {0, 1}表示第一个点，从起点到这个点的距离是0
    heap.push(make_pair(0, 1));

    while (heap.size())
    {
        pii t = heap.top();
        heap.pop();

        int ver = t.yy, distance = t.xx;

        // printf("ver = %d, distance = %d\n", ver, distance);

        if (st[ver]) continue; // 如果这个已经走到过了，说明存储的是冗余的数据
        st[ver] = true;

        // 更新最短路
        for (int i = h[ver]; i != -1; i = ne[i])
        {
            // 得到这条边指向哪里
            int j = e[i];
            if (dist[j] > dist[ver] + w[i])
            {
                // 如果从ver走向j的距离比直接到j的距离短，更新距离。
                dist[j] = dist[ver] + w[i]; // 这里注意是w[i]，因为w的下标是idx。
                // printf("dist[%d] = dist[%d] + w[%d] = %d = %d + %d\n", j,
ver, i, dist[j], dist[ver], w[i]);
                heap.push(make_pair(dist[j], j));
            }
```

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

int main()
{
    memset(h, -1, sizeof h);

    cin >> n >> m;

    while (m -- )
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
    }

    cout << dijkstra();
    return 0;
}
```

bellman-ford

给定一个n个点m条边的有向图，图中可能存在重边和自环， **边权可能为负数。**

请你求出从1号点到n号点的最多经过k条边的最短距离，如果无法从1号点走到n号点，输出impossible。

注意：图中可能 **存在负权回路** 。

输入格式

第一行包含三个整数n，m，k。

接下来m行，每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。

输出格式

输出一个整数，表示从1号点到n号点的最多经过k条边的最短距离。

如果不存在满足条件的路径，则输出“impossible”。

数据范围

$1 \le n, k \le 500$,
$1 \le m \le 10000$,
任意边长的绝对值不超过10000。

输入样例：

```
3 3 1
1 2 1
2 3 1
1 3 3
```

输出样例：

```
3
```

- 如果能求出最短路，那么这个图里面不能有负权回路。不然只要不断循环走这个回路，路径的长度就能无限减少。
- 如果用 `SPFA` 则要求图中不存在负环。
- 如果规定了**走的边数的个数**，那么不能用 `SPFA` ，只能用 `bellman-ford` 。而且如果规定了走的边数的个数，那么就算图中有负环也没关系了。（实际问题中比如说要求飞机的航线，但是换乘次数不能过多，即规定了走的边数的个数。）
- 因为 `bellman-ford` 算法要遍历到每一条边，所以不需要用邻接表或者邻接矩阵来存储，直接用一个结构体就可以。

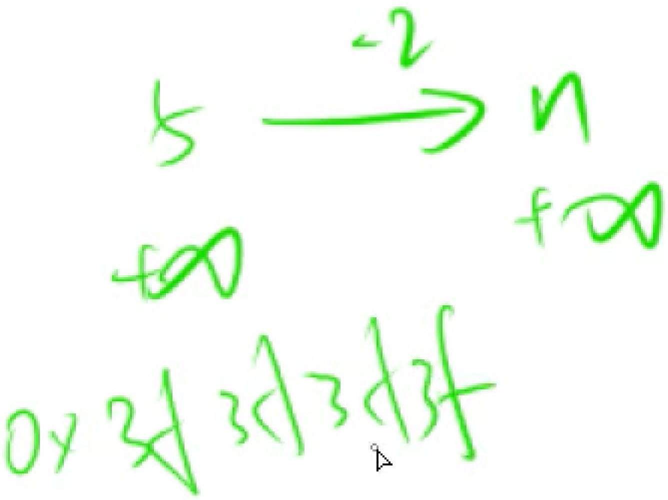
  ```
  struct
  {
      int a, b, c;
  } edges[N];
  ```

~~~~
+ 为了防止更新距离的时候出现串联的情况，要开一个 `int backup[N]` 数据，每一次更新数据的时候通过 `backup` 数据中存储的上一次的距离来更新。这样就不会出现串联的问题。（第三章 搜索与图论（二）1：25：00）

+ 判断能否到达的时候不能像 `dijkstra` 那样通过

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

来更新。因为有可能负权边会把正无穷给更新了



这样的话就不是严格等于正无穷了。所以对于 `bellman-ford` 算法，要使用

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

判断是否能到达。

- 如果有负环其更新的形式会是怎么样的还是不太理解。应该是题目限制了只能走k条边，所以算法里面是

  ```cpp
  for (int i = 0; i < k; i ++ )
  ```

- 题目代码如下

  ```cpp
  #include <iostream>
  #include <stdio.h>
  #include <string.h>
  #include <algorithm>

  using namespace std;

  const int N = 510, M = 10010;
  int n, m, k;
  int dist[N];
  int backup[N];

  struct
  {
      int a, b, w;
  ```

```
} edges[M];

void bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);

    // 初始化距离
    dist[1] = 0;

    // 一共只能走k条边。
    for (int i = 0; i < k; i ++ )
    {
        memcpy(backup, dist, sizeof dist);
        for (int j = 0; j < m; j ++ )
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            // 松弛操作，更新最短路径，如果从a到b比b当前的距离要短，则更新b的最短路径
            // 这里不能是min(backup[b], backup[a] + w)，因为这个for循环会一直更新
dist，如果使用backup就会出现问题。
            dist[b] = min(dist[b], backup[a] + w);
        }
    }
}

int main()
{
    cin >> n >> m >> k;
    for (int i = 0; i < m; i ++ )
    {
        int a, b, w;
        cin >> a >> b >> w;
        edges[i] = {a, b, w};
    }

    bellman_ford();

    if (dist[n] > 0x3f3f3f3f / 2) printf("impossible");
    else printf("%d\n", dist[n]);
    return 0;
}
```

- 一般的最短路问题中都不会有负环。

SPFA

给定一个n个点m条边的有向图，图中可能存在重边和自环， **边权可能为负数。**

请你求出1号点到n号点的最短距离，如果无法从1号点走到n号点，则输出impossible。

数据保证不存在负权回路。

输入格式

第一行包含整数n和m。

接下来m行每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。

输出格式

输出一个整数，表示1号点到n号点的最短距离。

如果路径不存在，则输出"impossible"。

数据范围

$1 \le n, m \le 10^5$,
图中涉及边长绝对值均不超过10000。

输入样例：

```
3 3
1 2 5
2 3 -3
1 3 4
```

输出样例：

```
2
```

- `SPFA` 是对 `bellman-ford` 算法的优化，思想是只要有节点变小了，那么我们就把它放到队列里面去。这样就不用遍历全部了。也就是

  ```
  if (dist[b] > backup[a] + w)
  ```

  那么就将这个节点放到队列里面。当然如果当前队列里面已经有了这个元素，即 `st[i] == true` 那么它就一定会被更新到，就不用再入队。而如果这个元素出队了，即已经更新了一遍其他元素的最短路径，那么就将 `st[i] = false` ，让之后他可以重新入队。

- 代码如下

  ```
  #include <iostream>
  #include <stdio.h>
  #include <algorithm>
  #include <string.h>
  #include <queue>

  using namespace std;

  const int N = 100010;
  int e[N], w[N], ne[N], h[N], idx;
  ```

```
int n, m;
bool st[N];
int dist[N];

void add(int a, int b, int c)
{
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx ++ ;
}

int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

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

    while (q.size())
    {
        int 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] == false)
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return dist[n];
}

int main()
{
    memset(h, -1, sizeof h);

    scanf("%d%d", &n, &m);

    while (m -- )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }

   int t = spfa();
```

```
    if (t == 0x3f3f3f3f) puts("impossible");
    else cout << t;

     return 0;
}
```

- 笔试题交 `SPFA` 一般都没事，一般数据比较水。

SPFA判负环

给定一个n个点m条边的有向图，图中可能存在重边和自环， **边权可能为负数。**

请你判断图中是否存在负权回路。

输入格式

第一行包含整数n和m。

接下来m行每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。

输出格式

如果图中**存在**负权回路，则输出“Yes”，否则输出“No”。

数据范围

$1 \le n \le 2000,$
$1 \le m \le 10000,$
图中涉及边长绝对值均不超过10000。

输入样例：

```
3 3
1 2 -1
2 3 4
3 1 -4
```

输出样例：

```
Yes
```

- dist[x] = dist[t] + w[i];

  cnt[x] = cnt[t] + 1;

  cnt[x] ≥ n

每次更新距离的时候同时也更新从1到这个点经过的边数 `cnt[x]` 。而如果 `cnt[x] >= n` ，则说明其至少经过了 `n + 1` 个点才到达这个点。说明路径中出现了环。

- 代码如下

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
#include <queue>

using namespace std;

const int N = 10010;

int n, m;
int dist[N];
bool st[N];
int cnt[N];
int e[N], ne[N], w[N], idx, h[N];

void add(int a, int b, int c)
{
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx ++ ;
}

bool spfa()
{
    queue<int> q;

    for (int i = 1; i <= n; i ++ )
    {
        st[i] = true;
        q.push(i);
    }

    while (q.size())
    {
        int 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 (st[j] == false)
                {
                    q.push(j);
                    st[j] = true;
                }
                if (cnt[j] >= n) return true;
```

```
            }
        }
    }

    return false;
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    while (m -- )
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
    }
    if (spfa()) puts("Yes");
    else puts("No");
    return 0;
}
```

floyd算法

给定一个n个点m条边的有向图，图中可能存在重边和自环，边权可能为负数。

再给定k个询问，每个询问包含两个整数x和y，表示查询从点x到点y的最短距离，如果路径不存在，则输出"impossible"。

数据保证图中不存在负权回路。

输入格式

第一行包含三个整数n，m，k

接下来m行，每行包含三个整数x，y，z，表示存在一条从点x到点y的有向边，边长为z。

接下来k行，每行包含两个整数x，y，表示询问点x到点y的最短距离。

输出格式

共k行，每行输出一个整数，表示询问的结果，若询问两点间不存在路径，则输出"impossible"。

数据范围

$1 \le n \le 200$,
$1 \le k \le n^2$
$1 \le m \le 20000$,
图中涉及边长绝对值均不超过10000。

输入样例：

```
3 3 2
1 2 1
2 3 2
1 3 1
2 1
1 3
```

输出样例：

```
impossible
1
```

- `floyd` 算法可以处理负权边，但是也是不能处理 **负权回路** 的。
- `floyd` 算法复杂度 `O(n ^ 3)` ，三重循环。
- 三重循环中通过

  ```
  g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
  ```

  不断更新最短路径。将从 `i` 到 `j` 的最短路径更新为从 `i` 到 `k` 然后再从 `k` 到 `j` 两者中的最短路径。
- 出现 `runtime error` 可以通过删代码的方法来判断问题出现在哪里。
- `floyd` 算法也存在和 `bellman-ford` 算法一样的问题，不能用

  ```
  if (d[a][b] == INF)
  ```

  来判断不通，因为有可能其距离会被负权边更新。所以要用

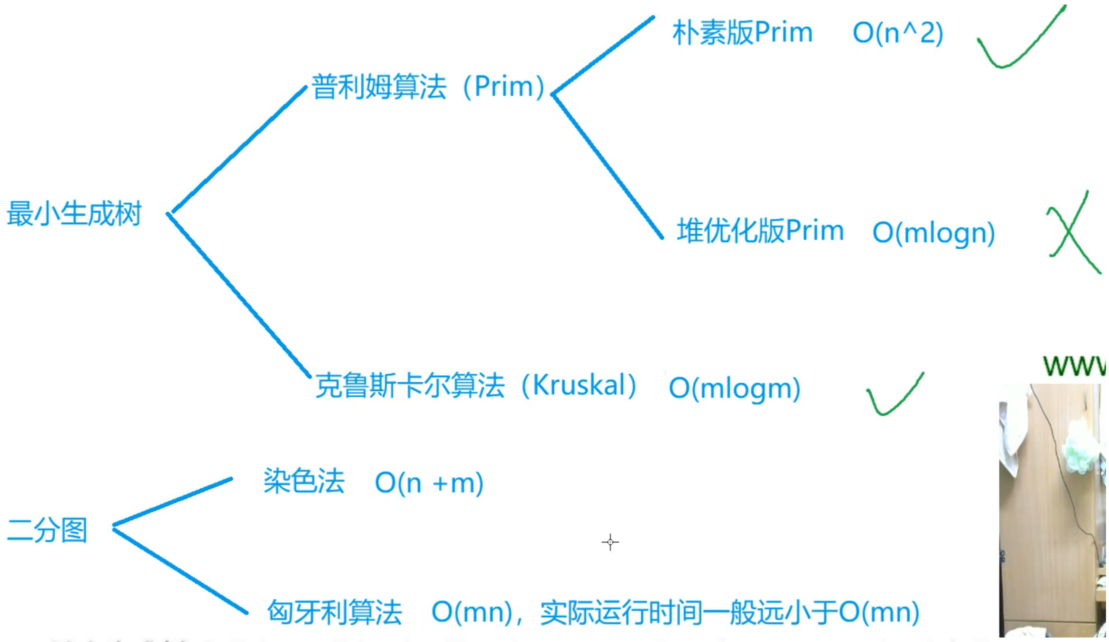
  ```
  if (d[a][b] > INF / 2)
  ```

  来替代。

- 代码如下

```
#include <iostream>
#include <stdio.h>
#include <algorithm>
#include <string.h>

using namespace std;

const int N = 220, INF = 1E9;
int g[N][N];
int n, m, T;

void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
            {
                // 判断从i到j的距离跟先从i到k然后从k到j的距离哪个比较短
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
            }
}

int main()
{
    cin >> n >> m >> T;
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) g[i][j] = 0; // 自己到自己的距离是0
            else g[i][j] = INF;

    while (m -- )
    {
        int x, y, z;
        cin >> x >> y >> z;
        g[x][y] = min(g[x][y], z);
    }

    floyd();

    while (T -- )
    {
        int x, y;
        cin >> x >> y;
        if (g[x][y] > INF / 2) puts("impossible");
        else cout << g[x][y] << endl;
    }

    return 0;
}
```

(三)

- 

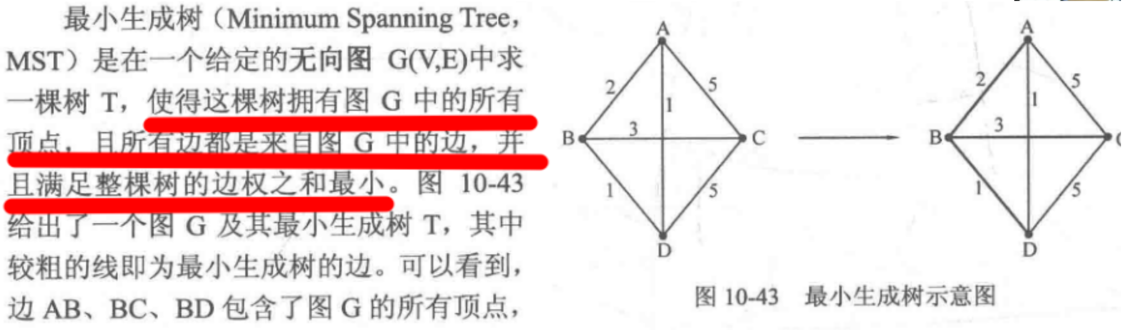
- 
最小生成树（Minimum Spanning Tree，MST）是在一个给定的无向图 G(V,E)中求一棵树 T，使得这棵树拥有图 G 中的所有顶点，且所有边都是来自图 G 中的边，并且满足整棵树的边权之和最小。图 10-43 给出了一个图 G 及其最小生成树 T，其中较粗的线即为最小生成树的边。可以看到，边 AB、BC、BD 包含了图 G 的所有顶点，

图 10-43 最小生成树示意图

prim

给定一个n个点m条边的无向图，图中可能存在重边和自环，边权可能为负数。

求最小生成树的树边权重之和，如果最小生成树不存在则输出impossible。

给定一张边带权的无向图G=(V, E)，其中V表示图中点的集合，E表示图中边的集合，n=|V|，m=|E|。

由V中的全部n个顶点和E中n-1条边构成的无向连通子图被称为G的一棵生成树，其中边的权值之和最小的生成树被称为无向图G的最小生成树。

输入格式

第一行包含两个整数n和m。

接下来m行，每行包含三个整数u，v，w，表示点u和点v之间存在一条权值为w的边。

输出格式

共一行，若存在最小生成树，则输出一个整数，表示最小生成树的树边权重之和，如果最小生成树不存在则输出impossible。

数据范围

$1 \le n \le 500$,
$1 \le m \le 10^5$,
图中涉及边的边权的绝对值均不超过10000。

输入样例：

```
4 5
1 2 1
1 3 2
1 4 3
2 3 2
3 4 4
```

输出样例：

```
6
```

- 求最小生成树的应用：比如有许多个城市，每两个城市之间都要铺设公路，问应该怎么铺设公路最短。（任意两条公路相交也没关系）
- 当所有点不连通的时候不存在最小生成树。
- `prim` 中的 `dist[t]` 表示的是**这个点到集合的距离**，而 `dijkstra` 算法中的 `dist[t]` 表示的是这个点到起点的距离。所以在 `prim` 算法中每一次找到新的点 **t点** 之后就要将整个图里面所有的点到集合的距离更新。即 `dist[j] = min(dist[j], g[t][j])` 。

```
for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
```

- 代码如下

```
#include <iostream>
#include <stdio.h>
#include <algorithm>
#include <string.h>
#include <stdlib.h>

using namespace std;

const int N = 510, INF = 0x3f3f3f3f;
int g[N][N], dist[N];
bool st[N];
int n, m;
```

```
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 ++ )
            // 如果还没有找到点或者找到了离生成树更近的点
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 如果不是第一次遍历，而且没有任何一个点可以到达这个图，说明图不连通
        if (i && dist[t] == INF) return INF;

        // 如果不是第一次遍历（dist[t] != INF）
        // 这里一定要先res+=dist[t]再更新距离，因为如果有自环会被自己更新，所以要提前更
新
        if (i) res += dist[t];
        st[t] = true;

        // 和dij不太一样，具体看笔记第三条
        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}

int main()
{
    // 为什么这里不把自己到自己的距离g[i][i]设为0，不太理解
    memset(g, 0x3f, sizeof g);
    cin >> n >> m;
    for (int i = 0; i < m; i ++ )
    {
        int a, b, c;
        cin >> a >> b >> c;
        // 因为是无向图，所以双向的边都要更新
        g[a][b] = g[b][a] = min(g[a][b], c);
    }

    int t = prim();
    if (t == INF) puts("impossible");
    else cout << t;
    return 0;
}
```

kruskal算法

给定一个n个点m条边的无向图，图中可能存在重边和自环，边权可能为负数。

求最小生成树的树边权重之和，如果最小生成树不存在则输出impossible。

给定一张边带权的无向图G=(V, E)，其中V表示图中点的集合，E表示图中边的集合，n=|V|，m=|E|。

由V中的全部n个顶点和E中n-1条边构成的无向连通子图被称为G的一棵生成树，其中边的权值之和最小的生成树被称为无向图G的最小生成树。

输入格式

第一行包含两个整数n和m。

接下来m行，每行包含三个整数u，v，w，表示点u和点v之间存在一条权值为w的边。

输出格式

共一行，若存在最小生成树，则输出一个整数，表示最小生成树的树边权重之和，如果最小生成树不存在则输出impossible。

数据范围

$1 \le n \le 10^5$,
$1 \le m \le 2 * 10^5$,
图中涉及边的边权的绝对值均不超过1000。

输入样例：

```
4 5
1 2 1
1 3 2
1 4 3
2 3 2
3 4 4
```

输出样例：

```
6
```

- Kruskal 算法

  ① 将所有边按权重从小到大排序 O(mlogm)

  ② 枚举每条边a,b 权重c
  if a,b不连通
  将这条边加入集合中 O(m)

- 要先排序一遍，这也是整个算法最慢的部分，时间复杂度是 `O(mlogm)`
- 加入集合的思想和并查集类似。
- 结构体排序的时候重载小于号。
- `kruskal` 就是 `sort` 和 **并查集结合在一起**。

```cpp
#include <iostream>
#include <stdio.h>
#include <algorithm>
#include <string.h>

using namespace std;

const int N = 200010;
int p[N];
int n, m;

struct Edge
{
    int a, b, w;
    bool operator< (const Edge &W) const
    {
        return w < W.w;
    }
} edges[N];

int find(int x)
{
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < m; i ++ )
    {
        int a, b, w;
        cin >> a >> b >> w;
        edges[i] = {a, b, w};
    }
    sort(edges, edges + m);

    // res用来存路径长度，cnt用来存这棵最小生成树里面有多少个元素
    int res = 0, cnt = 0;

    for (int i = 0; i <= n; i ++ ) p[i] = i;

    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        int fa = find(a), fb = find(b);
        if (fa != fb)
        {
            res += w;
            cnt ++ ;
            p[fa] = fb;
        }
    }
    if (cnt < n - 1) puts("impossible");
    else cout << res << endl;
    return 0;
}
```

二分图

给定一个n个点m条边的无向图，图中可能存在重边和自环。

请你判断这个图是否是二分图。

输入格式

第一行包含两个整数n和m。

接下来m行，每行包含两个整数u和v，表示点u和点v之间存在一条边。

输出格式

如果给定图是二分图，则输出“Yes”，否则输出“No”。

数据范围

$1 \leq n, m \leq 10^5$

输入样例：

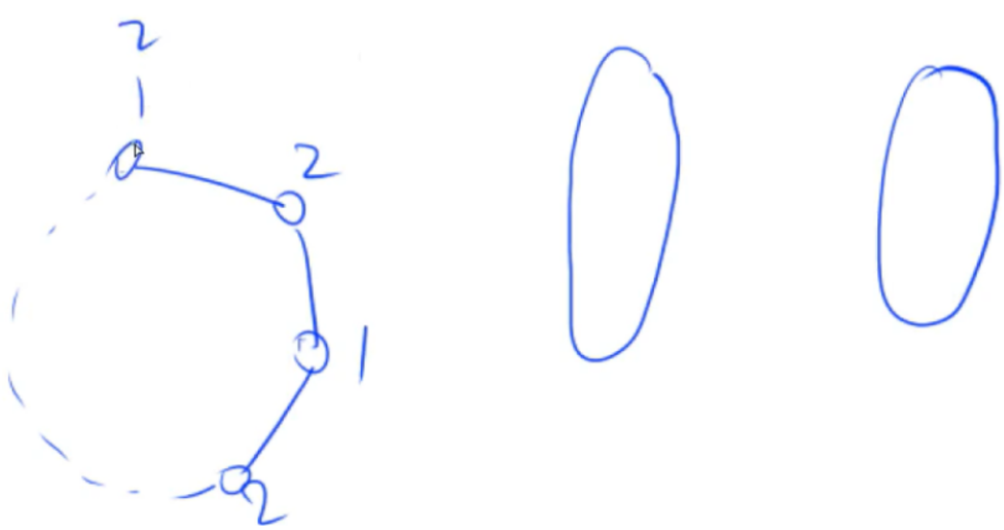
```
4 4
1 3
1 4
2 3
2 4
```

输出样例：

```
Yes
```

- 二分图当且仅当图中不含奇数环
- 二分图的定义就是我们可以把所有点划分到两边，使得 **所有的边都是在集合之间的，集合内部没有边。**

二分图当且仅当图中不含奇数环



- 如果是二分图则一定没有奇数环，反过来也成立，如果没有奇数环，那么这个图一定是二分图。

- 染色法判二分图 `O(nm)` 代码如下

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

const int N = 100010, M = 200010;

int e[M], ne[M], color[N], h[N], idx;
int n, m;

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

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])
        {
            if (!dfs(j, 3 - c))
                return false;
        }
        else if (color[j] == c)
            return false;
    }

    return true;
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for (int i = 0; i < m; i ++ )
    {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }

    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        // 如果这个点没有被染过色
        if (!color[i])
        {
            if (!dfs(i, 1))
            {
                flag = false;
                break;
```

```
            }
        }

    if (flag) puts("Yes");
    else puts("No");

    return 0;
}
```

匈牙利算法

给定一个二分图，其中左半部包含n_1个点（编号1~n_1），右半部包含n_2个点（编号1~n_2），二分图共包含m条边。

数据保证任意一条边的两个端点都不可能在同一部分中。

请你求出二分图的最大匹配数。

> 二分图的匹配：给定一个二分图G，在G的一个子图M中，M的边集{E}中的任意两条边都不依附于同一个顶点，则称M是一个匹配。
>
> 二分图的最大匹配：所有匹配中包含边数最多的一组匹配被称为二分图的最大匹配，其边数即为最大匹配数。

输入格式

第一行包含三个整数 n_1、 n_2 和 m。

接下来m行，每行包含两个整数u和v，表示左半部点集中的点u和右半部点集中的点v之间存在一条边。

输出格式

输出一个整数，表示二分图的最大匹配数。

数据范围

$1 \le n_1, n_2 \le 500$,
$1 \le u \le n_1$,
$1 \le v \le n_2$,
$1 \le m \le 10^5$

输入样例:

```
2 2 4
1 1
1 2
2 1
2 2
```

输出样例:

```
2
```

- 递归，如果找到的女生名花有主就看她能不能换个老公

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;
```

```cpp
const int N = 520, M = 100010;

int e[M], ne[M], h[N], idx, n1, n2, m;
bool st[N];
int match[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

bool find(int x)
{
    for (int i = h[x]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (!match[j] || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n1 >> n2 >> m;
    while (m -- )
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
    }

    int res = 0;

    for (int i = 1; i <= n1; i ++ )
    {
        memset(st, false, sizeof st);
        if (find(i)) res ++ ;
    }

    cout << res << endl;

    return 0;
}
```

数论

试除法判断质数

- 试除法的时间复杂度 `O(sqrt(n))` 。（一定根号n，并不是说最坏根号n）。

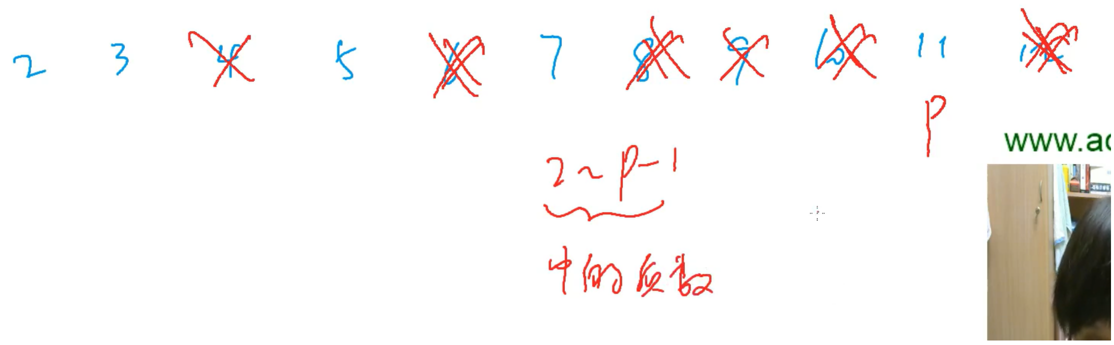
不要写 `i * i <= n` 这样容易溢出。推荐写法是 `i <= n / i`

试除法分解质因数

- 最坏 `O(sqrt(n))` 但是一般没那么慢。
- 这里会枚举所有的数而不是所有的质数，但并不会影响结果。因为当我们枚举到 `i` 的时候说明 `x` 中已经不包含任何小于等于 `i` 的因子了。所以最终的结果不会有问题。

```
#include <iostream>
#include <stdio.h>
#include <algorithm>

using namespace std;

void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
    {
        if (x % i == 0) // 只要这个成立，i一定是质数。
        {
            int s = 0;
            while (x % i == 0)
            {
                s ++ ;
                x /= i;
            }
            cout << i << " " << s << endl;
        }
    }
    if (x > 1) cout << x << " 1" << endl;
    puts("");
}

int main()
{
    int n;
    cin >> n;
    while (n -- )
    {
        int t;
        cin >> t;
        divide(t);
    }

    return 0;
}
```

筛质数

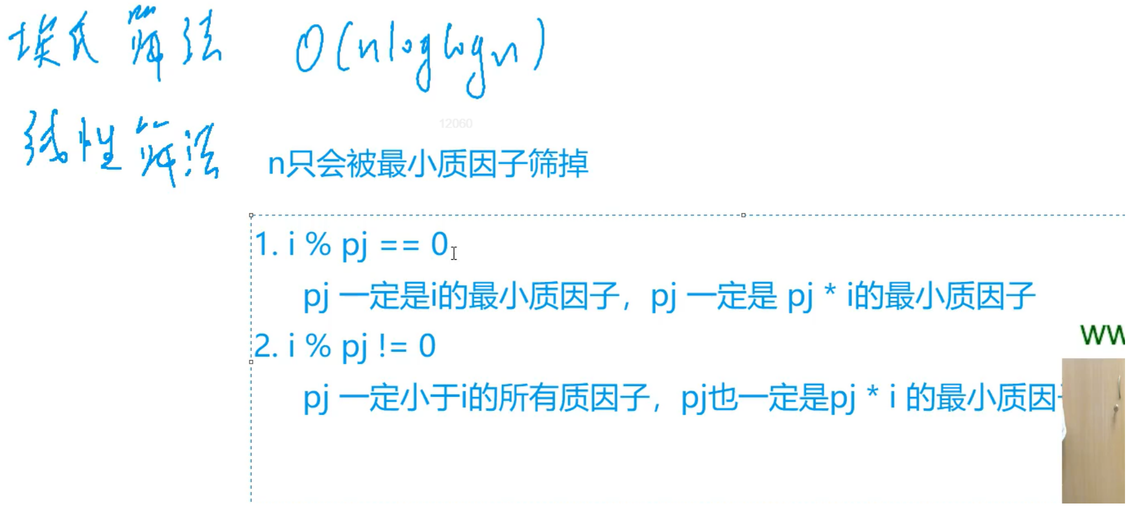
- 

  当一个数不是质数的时候就不需要筛掉其所有的倍数，只需要筛掉 `p` 的质数次方的那些数字就可以了。

- 朴素筛法时间复杂度是 `O(nlogn)` （logn以e为底）
- 优化后的筛法（埃筛）时间复杂度

  跟 `O(n)` 是一个数量级的。

- 

- 埃筛代码如下

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

const int N = 1000010;

int primes[N], cnt;
bool st[N];

void get_primes(int n)
{
```

```
    for (int i = 2; i <= n; i ++ )
    {
        if (st[i]) continue; // 如果是合数（已经被筛过了），跳过
        // 否则说明这个数是质数，开始用这个数筛
        else
        {
            primes[cnt ++ ] = i;
            // 只有当这个数是质数的时候才用这个数筛，放在else里面
            for (int j = i + i; j <= n; j += i)
                st[j] = true;
        }
    }
}

int main()
{
    int n;
    cin >> n;

    get_primes(n);

    cout << cnt << endl;

    return 0;
}
```

- 线性筛法代码如下
- 其原理是保证了每个合数都是被其 **最小质因子** 筛掉的。

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

const int N = 1000010;

int primes[N], cnt;
bool st[N];

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        // 循环的判断条件不用加上j < cnt，因为如果这个数是质数的话一定会在primes[j] = i的时候停下来（因为已经加进去了），而如果是合数的话会在其最小质因子处停下来（因为其最小质因子一定小于其本身）。
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break; // 找到了最小质因子primes[j]
        }
    }
}
```

```
int main()
{
    int n;
    cin >> n;

    get_primes(n);

    cout << cnt << endl;

    return 0;
}
```

试除法求约数

- 时间复杂度是 `O(sqrt(n))` 。

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

vector<int> get_divisor(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
    {
        if (x % i == 0)
        {
            res.push_back(i);
            // 防止 i * i == x 的时候被推进去两次
            if (i != x / i) res.push_back(x / i);
        }
    }
    sort(res.begin(), res.end());
    return res;
}

int main()
{
    int T;
    cin >> T;
    while (T -- )
    {
        int x;
        cin >> x;
        auto res = get_divisor(x);
        for (auto a : res) cout << a << " ";
        puts("");
    }

    return 0;
}
```

约数个数 && 约数之和

- 在 `INT_MAX` 范围内约数个数最多大概在 **1500** 左右。
- 2. 约数

  (1) 试除法求一个数的所有约数

  (2) 约数个数 $(\alpha_1+1)(\alpha_2+1)\cdots(\alpha_k+1)$

  (3) 约数之和 $(p_1^0+p_1^1+\dots+p_1^{\alpha_1})\cdots(p_k^0+p_k^1+\dots+p_k^{\alpha_k})$

- 先对于每一个数分解质因数，得到 `a1` 、 `a2` 等等，然后套 `(a1 + 1) * (a2 + 1) * (a3 + 1) ...` 的公式。

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <map>
#include <unordered_map>

using namespace std;

const int mod = 1e9 + 7;
typedef long long ll;

int main()
{
    // 存放a1、a2 ...
    unordered_map<int, int> primes;
    int T;
    cin >> T;
    while (T -- )
    {
        int x;
        cin >> x;
        // 试除法分解因数
        for (int i = 2; i <= x / i; i ++ )
        {
            while (x % i == 0)
            {
                x /= i;
                primes[i] ++ ;
            }
        }

        if (x > 1) primes[x] ++ ;
    }

    ll res = 1;
    for (auto prime : primes) res = res * (prime.second + 1) % mod;

    cout << res << endl;

    return 0;
```

```
}
```

- 约数之和（这里 `t * p + 1`）不太理解

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <map>
#include <unordered_map>

using namespace std;

const int mod = 1e9 + 7;
typedef long long ll;

int main()
{
    // 存放a1、a2 ...
    unordered_map<int, int> primes;
    int T;
    cin >> T;
    while (T -- )
    {
        int x;
        cin >> x;
        // 试除法分解因数
        for (int i = 2; i <= x / i; i ++ )
        {
            while (x % i == 0)
            {
                x /= i;
                primes[i] ++ ;
            }
        }

        if (x > 1) primes[x] ++ ;
    }

    ll res = 1;

    for (auto prime : primes)
    {
        int p = prime.first, a = prime.second;
        ll t = 1;
        while (a -- ) t = (t * p + 1) % mod;
        res = res * t % mod;
    }

    cout << res << endl;

    return 0;
}
```