NTUT_King ICPC Team Notebook

Contents

1 基礎

1.1 關鍵字思考

一些寫題目會用到的小技巧：排容原理、二分搜尋、雙向搜尋、塗色問題、貪心、位元運算、暴力搜尋、

1.2 C++ 基礎

```cpp
// * define int long long 避免溢位問題
// * cin、cout 在測資過多時最好加速
// * define debug 用來測試

#include <bits/stdc++.h>
#define int long long
#define debug

using namespace std;

main()
{
    #ifdef debug
    freopen("in1.txt", "r", stdin);
    freopen("out1.txt", "w", stdout);
    #endif // debug
    // 讀寫加速
    // 關閉iostream 物件和cstdio 流同步以提高輸入輸出的效率
    ios::sync_with_stdio(false);
    // 可以通過tie(0)（0表示NULL）來解除cin 與cout 的繫結，進一步加快執行效率
    cin.tie(0);
}
```

1.3 C++ 易忘的內建函數

```
## 易忘的內建函數
### 輸入輸出
* gets(char*)
* sscanf(char*, "%d:%d:%d %lf", &h, &m, &s, &speed_new)
%d 表示 int
%lf 表示 double
* printf("%.2d:%.2d:%.2d %.2lf km\n",h,m,s,din)
.2 表示保留 2 位小數(%d 是整數，會自動捨去小數)
### 字串處理
* string.length() 輸出字串長度
* string.substr(start, len) 輸出從 start 開始，長度為 len 的字串
* string.find(string) 尋找字串位置
### 資料型別
* string = to_string(int)
* int = atoi(string.c_str())
### 運算
* lower_bound(begin, end, num)
    * 從陣列的 begin 位置到 end - 1 位置二分查詢第一個大於或等於 num 的數字，找到返回該數字的地址，不存在則返回 end
    * 通過返回的地址減去起始地址，得到找到數字在陣列中的下標 begin
* __builtin_popcount(int)
    * 回傳整數轉成二進值時所包含 1 的數量
* ^
    * 互斥或 xor 運算子
```

1.4 python 常用

```python
# ## Python 內建大數
#  可以直接用int() 和各個運算子計算
#  雖然Python 有BigInt()，但用不到

from sys import stdin, stdout

def main():
    n = int(stdin.readline())
    for i in range(n):
        line = stdin.readline().split("/")
        # 可直接轉換成大數
        p = int(line[0])
        q = int(line[1])

        # 求最大公因數
        gcdNum = gcd(p, q)

        stdout.write(str(gcdNum))
        stdout.write("\n")

main()
```

```
# ## 字串處理
# * string[start :  end] 取start   end - 1 的字串
# * string.find(string) 尋找字串位置

# ## 數學函式
# * round(number) 四捨五入


# ## 易錯事項
# * / 除法運算，結果總是返回浮點型別
# * // 取整除，結果返回捨去小數部分的整數
# * stdout.write(str(p)) 不能沒有str()
# * write 只能輸出字串
# * 測試時取值只能用以下程式(Spyder、Jupyter 實測)
input(string)
print(..., end = '')

# * 上傳程式時取值能用以下程式(Online Judge 實測可以Accepted)
input(string)
print(..., end = '')
stdin.readline(...)
stdout.write(...)

# * 用try 接受多行輸入
def main():
    # 用try 接受多行輸入
    try:
        while(True):
            # 輸出對應的n
            n = int(input(""))
            print(n)
    except:
        # 沒輸入內容可直接跳過
        pass

main()
```

2 承恩-數論

2.1 約瑟夫斯-每兩個殺一次

```
#include <iostream>
#include <math.h>
using namespace std;
int T,n,k,kase;
int josephus(){
    k=0;//因為2的0次方等於1
    while(pow(2,k)<n){
        k+=1;
    }
    if(pow(2,k)==n){
        return 1;//如果n剛好是2的次方
    }
    else{
        return 2*(n-pow(2,k-1))+1;//否則回傳2b+1
    }
}
int main()
{
    cin>>T;
    while(T--){
        cin>>n;
        int repeat=0;
        while(n!=josephus()){
            repeat++;
            n=josephus();
        }
        cout<<"Case "<<++kase<<": "<<repeat<<" "<<n<<endl;
    }
    return 0;
}
```

2.2 約瑟夫斯-一般情況

```
//f(N,M)=(f(N1,M)+M)//f(N,M)表示，N人，每到M掉那人，最胜利者的
//f(N1,M)表示，N-1人，每到M掉那人，最胜利者的
#include <iostream>

using namespace std;
int T,n,k,kase;
int main()
{
    cin>>T;
    while (T--){
        cin>>n>>k;
        int survivor=0;
        for(int i =2;i<=n;i++){
            survivor = (survivor+k)%i;
        }
        survivor++;
        cout<<"Case "<<++kase<<": "<<survivor<<endl;
    }
    return 0;
}
```

2.3 最大子數列

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

using namespace std;
int X[100050];
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    string str;
    while(getline(cin,str)){
        istringstream temp(str);
        int i=0,sum=0,max=0;
        while(temp>>X[i++]);
        for(int j=0;j<i-1;j++){
            sum+=X[j];//將數值加進來
            if(max<sum){//如果當前數值總和>max，就儲存
                max=sum;
            }
            if(sum<0){
                sum=0;
            }
        }
        cout<<max<<'\n';
    }
    return 0;
}
```

3 建榮-演算法

3.1 深度優先搜尋

```
```c=
#include <iostream>
using namespace std;

int visited[7];
int map[7][7] ;
void dfs(int id){
    cout << id << ' ';
    for (int i = 1; i <= 6; i++){
        if(map[id][i] == 1 && visited[i] == 0){
            visited[i] = 1;
            dfs(i);
        }
    }
}
int main(void){
    visited[1] = 1;
    dfs(1);
}
```
```

3.2 廣度優先搜尋

```cpp=
#include <iostream>
#include <queue>

using namespace std;

queue<int> q;
int visited[7];
int map[7][7]

int main(void){
    visited[1] = 1;
    q.push(1);
    while(q.size() > 0){
        int id = q.front();
        cout << id << ' ';
        for (int i = 1; i <= 6; i++){
            if(map[id][i] == 1 && visited[i] == 0){
                visited[i] = 1;
                q.push(i);
            }
        }
        q.pop();
    }
}
```

3.3 二分搜

```c=
#include <iostream>

using namespace std;

int A[10] = {0, 3, 5, 6, 8, 9, 12, 14, 25, 30};

int bsearch(int num, int l, int r){
    int m = (l + r) / 2;
    if(num == A[m])
        return m;
    if(num > A[m])
        return bsearch(num, m + 1, r);
    return bsearch(num, l, m);
}

int main(void){
    cout << bsearch(9,0,9);
}
```

3.4 二元數的走訪

```c=
#include <iostream>

using namespace std;

int btree[8] = {-1, 1, 2, 3, -1, 5, 6, 7};

void Preorder(int id){
    if(id > 7 || btree[id] == -1){
        return;
    }
    cout << id << ' ';
    Preorder(id * 2);
    Preorder(id * 2 + 1);
}

int main(void){
    Preorder(1);
}
```

3.5 最大公因數

```
---
tags: Algorithm
---
# GCD
```c
#include <iostream>

using namespace std;

int gcd(int a, int b){
    if(a==0)
        return b;
    if(b==0)
        return a;
    if(a<b){
        return gcd(a, b % a);
    }
    return gcd(a % b, b);
}

int main(void){
    cout << gcd(546, 429);
}
```

4 建榮-幾何

4.1 高中數學

// 高中數學統整
// * 餘式定理：多項式$f(x)$ 被一次式$ax+b$ 所除的餘式為$f(-\frac{b}{a})$
// * 等比數列：$S_n = \{na|r = 1, \frac{a_1(r^n-1)}{(r-1)}|r! = 1\}$
// * $sigma$公式：
// - $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
// - $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
// - $\sum_{k=3}^{n} k^3 = (\frac{(n(n+1)}{2})^2$
// * 重複組合$H_m^n = C_m^{n+m-1}$
// * 正弦定理$\frac{a}{sinA} = \frac{b}{sinB} = \frac{C}{sinC}$
// * 餘弦定理$cosA = \frac{b^2+c^2-a^2}{2bc}$

4.2 點的模板

點的模板

```c=
class Point {
    private:
        int _x, _y;
    public:
        Point(int x, int y) : _x(x), _y(y){};
        int getX() const { return _x; }
        int getY() const { return _y; }

        bool operator==(const Point& other_point) const {
            return _x == other_point.getX() && _y < other_point.getY();
        }
        Point& operator+(const Point& other_point) const {
            return *(new Point(_x + other_point.getX(), _y + other_point.getY()));
        }
        Point& operator-(const Point& other_point) const {
            return *(new Point(_x - other_point.getX(), _y - other_point.getY()));
        }
        int cross(const Point& other_point) {
            return _x * other_point.getY() - _y * other_point.getX();
        }
};
```

4.3 向量計算
```

```cpp
#include <iostream>

using namespace std;

struct Point {float x, y;}; // 點的資料結構
typedef Point Vector;       // 向量的資料結構，和點一樣

// 向量的長度
float length(Vector v)
{
    return sqrt(v1.x * v1.x + v2.y * v2.y);
// return sqrt(dot(v, v));
}
void base_height(Point p, Point p1, Point p2)
{
    Vector v1 = p1 - p;
    Vector v2 = p2 - p;

    float base = fabs(dot(v1, v2)) / length(v1);
    float height = fabs(cross(v1, v2)) / length(v1);
}
// 向量oa與向量ob進行內積，判斷∠aob之大小。
//內積大於0 時，兩向量夾角小於90；等於0 時，夾角等於90；小於零時，夾角大於90且小於180
float dot(Point o, Point a, Point b)
{
    return (a.x-o.x) * (b.x-o.x) + (a.y-o.y) * (b.y-o.y);
}
// 向量oa與向量ob進行外積，判斷oa到ob的旋轉方向。
//外積大於0 時，兩向量前後順序為逆時針順序（在180之內）；等於0 時，兩向量平行，也就是指夾角等於0或180；小於0 時，兩向量
      前後順序為順時針順序（在180之內）。
float cross(Point o, Point a, Point b)
{
    return (a.x-o.x) * (b.y-o.y) - (a.y-o.y) * (b.x-o.x);
}
void θsin_cos_(Point p, Point p1, Point p2)
{
    Point p, p1, p2;

    Vector v1 = p1 - p;
    Vector v2 = p2 - p;

    float l1 = length(v1);
    float l2 = length(v2);

    float θcos = dot(v1, v2)    / l1 / l2;
    float θsin = cross(v1, v2) / l1 / l2;

    float θ  = acos(θcos);    // [0, π]
    float θ  = asin(θsin);    // [-π/2, π/2]
}
```

4.4 直線模板

```cpp
# 直線模板
```c=
class Line {
    private:
        Point _p1, _p2;
    public:
        Line(Point p1, Point p2) : _p1(p1), _p2(p2){};
        Point Point1() const {
            return _p1;
        }
        Point Point2() const {
            return _p2;
        }
        bool isIntersect(const Line& other_line) const {
            int max_other_x = max(other_line.Point1().getX(), other_line.Point2().getX());
            int max_other_y = max(other_line.Point1().getY(), other_line.Point2().getY());
            int min_other_x = min(other_line.Point1().getX(), other_line.Point2().getX());
            int min_other_y = min(other_line.Point1().getY(), other_line.Point2().getY());
            int max_self_x = max(_p1.getX(), _p2.getX());
            int max_self_y = max(_p1.getY(), _p2.getY());
            int min_self_x = min(_p1.getX(), _p2.getX());
            int min_self_y = min(_p1.getY(), _p2.getY());

            if ((max_self_x >= min_other_x) && (max_other_x >= min_self_x) && (max_self_y >=
                 min_other_y) && (max_other_y >= min_self_y)) {
                if ((_p1 - other_line.Point1()).cross(_p1 - _p2) * (_p1 - other_line.Point2()).cross(
                     _p1 - _p2) <= 0) {
                    if ((other_line.Point1() - _p1).cross(other_line.Point1() - other_line.Point2()) *
                         (other_line.Point1() - _p2).cross(other_line.Point1() - other_line.Point2()
                        ) <= 0) {
                        return true;
                    }
                }
            }
            return false;
        }
};
```
```

4.5 找三角形外心

```cpp
# 找三角形外心
```c=
#include <cmath>
#include <iostream>
using namespace std;

struct Point {
    double x;
    double y;
    Point() {}
    Point(double X, double Y) {
        x = X;
        y = Y;
    }
};

double distance_p2p(Point p1, Point p2) {
    return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

Point p[100];
int n;

class Circle {
    public:
        double r;
        Point c;
        Circle(Point p1, Point p2) : r(distance_p2p(p1, p2) / 2), c((p1.x + p2.x) / 2, (p1.y + p2.y) /
              2){}
        Circle(Point p1, Point p2, Point p3) {
            double A1 = p1.x - p2.x, B1 = p1.y - p2.y, C1 = (p1.x * p1.x - p2.x * p2.x + p1.y * p1.y -
                  p2.y * p2.y) / 2;
            double A2 = p3.x - p2.x, B2 = p3.y - p2.y, C2 = (p3.x * p3.x - p2.x * p2.x + p3.y * p3.y -
                  p2.y * p2.y) / 2;
            c.x = (C1 * B2 - C2 * B1) / (A1 * B2 - A2 * B1);
            c.y = (A1 * C2 - A2 * C1) / (A1 * B2 - A2 * B1);
            r = distance_p2p(c, p1);
        }
        Circle() {}
};
double find_smallest_r() {
    Circle c(p[0], p[1]);
    for (int i = 2; i < n; i++) {
        if (distance_p2p(c.c, p[i]) > c.r) {
            c = Circle(p[0], p[i]);
            for (int j = 1; j < i; j++) {
                if (distance_p2p(c.c, p[j]) > c.r) {
                c = Circle(p[j], p[i]);
                    for (int k = 0; k < j; k++) {
                        if (distance_p2p(c.c, p[k]) > c.r) {
                            c = Circle(p[j], p[i], p[k]);
                        }
                    }
                }
            }
        }
    }
    return c.r;
}
```
```

4.6 點在直線的上或下

```cpp
# 點在直線的上或下
```c=
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;
class Point {
    private:
        int _x, _y;
    public:
        Point(int x, int y) : _x(x), _y(y){};
```
```

```
        int getX() const { return _x; }
        int getY() const { return _y; }
        Point& operator-(const Point& other_point) const {
            return *(new Point(_x - other_point.getX(), _y - other_point.getY()));
        }
        int cross(const Point& other_point) {
            return _x * other_point.getY() - _y * other_point.getX();
        }
};
class Line {
    private:
        Point _p1, _p2;
    public:
        Line(Point p1, Point p2) : _p1(p1), _p2(p2){};
        Point Point1() const {
            return _p1;
        }
        Point Point2() const {
            return _p2;
        }
        bool isIntersect(const Line& other_line) const {
            int max_other_x = max(other_line.Point1().getX(), other_line.Point2().getX());
            int max_other_y = max(other_line.Point1().getY(), other_line.Point2().getY());
            int min_other_x = min(other_line.Point1().getX(), other_line.Point2().getX());
            int min_other_y = min(other_line.Point1().getY(), other_line.Point2().getY());
            int max_self_x = max(_p1.getX(), _p2.getX());
            int max_self_y = max(_p1.getY(), _p2.getY());
            int min_self_x = min(_p1.getX(), _p2.getX());
            int min_self_y = min(_p1.getY(), _p2.getY());

            if ((max_self_x >= min_other_x) && (max_other_x >= min_self_x) && (max_self_y >=
                min_other_y) && (max_other_y >= min_self_y)) {
                if ((_p1 - other_line.Point1()).cross(_p1 - _p2) * (_p1 - other_line.Point2()).cross(
                    _p1 - _p2) <= 0) {
                    if ((other_line.Point1() - _p1).cross(other_line.Point1() - other_line.Point2()) *
                        (other_line.Point1() - _p2).cross(other_line.Point1() - other_line.Point2()
                        ) <= 0) {
                        return true;
                    }
                }
            }
            return false;
        }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int x_s, y_s, x_e, y_e;
        cin >> x_s >> y_s >> x_e >> y_e;
        Line line(Point(x_s, y_s), Point(x_e, y_e));
        int x_1, x_2, y_1, y_2;
        cin >> x_1 >> y_1 >> x_2 >> y_2;
        int x_l = max(x_1, x_2), x_r = min(x_1, x_2), y_t = max(y_1, y_2), y_b = min(y_1, y_2);
        if (x_s < x_l && x_e < x_l && x_s > x_r && x_e > x_r && y_s < y_t && y_e < y_t && y_s > y_b &&
            y_e > y_b) {
            cout << "T" << endl;
        } else {
            Point left_top(x_l, y_t);
            Point right_top(x_r, y_t);
            Point left_bottom(x_l, y_b);
            Point right_bottom(x_r, y_b);
            Line left(left_top, left_bottom);
            Line right(right_top, right_bottom);
            Line top(left_top, right_top);
            Line bottom(left_bottom, right_bottom);
            if (left.isIntersect(line) || right.isIntersect(line) || top.isIntersect(line) || bottom.
                isIntersect(line)) {
                cout << "T" << endl;
            } else {
                cout << "F" << endl;
            }
        }
    }
    return 0;
}
```

4.7 兩直線交點

```
# 兩直線交點
```c=
class Vector {
    private:
        double _x;
        double _y;

    public:
        Vector(double x, double y) : _x(x), _y(y) {}
        double cross(const Vector& other_vector) const {
            return _x * other_vector._y - _y * other_vector._x;
        }
        double dot(const Vector& other_vector) const {
            return _x * other_vector._x + _y * other_vector._y;
        }
        double getX() { return _x; }
        double getY() { return _y; }

        Vector operator*(double k) const {
            return *(new Vector(k * _x, k * _y));
        }
};
Vector findIntersectionVector(const Vector& a, const Vector& b, const Vector& u) {
    return a * (u.cross(b) / a.cross(b));
}
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    cout << "INTERSECTING LINES OUTPUT" << endl;

    for (int i = 0; i < n; i++) {
        double x1, y1, x2, y2, x3, y3, x4, y4;
        cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3 >> x4 >> y4;
        if ((x1 - x2) * (y3 - y4) == (x3 - x4) * (y1 - y2)) {
            if (Vector(x1 - x3, y1 - y3).cross(Vector(x1 - x2, y1 - y2)) == 0) {
                cout << "LINE" << endl;
            } else {
                cout << "NONE" << endl;
            }
        } else {
            Vector intersectionVector = findIntersectionVector(Vector(x2 - x1, y2 - y1), Vector(x4 -
                x3, y4 - y3), Vector(x2 - x4, y2 - y4));
            cout << "POINT " << fixed << setprecision(2) << x2 - intersectionVector.getX() << " " <<
                fixed << setprecision(2) << y2 - intersectionVector.getY() << endl;
        }
    }
    cout << "END OF OUTPUT" << endl;

    return 0;
}
```

5 大衛-動態規劃

5.1 背包問題

```
memset(dp, INF, sizeof(dp));
memset(dp[0], 0, sizeof(dp[0])); //車子是0 貨箱時，一定沒辦法買水果，因此最低價都是0

for(int i = 0; i <= 每種水果; i++){
    for(int j = 0; j <= 卡車容量; j++){
        for(int k = 0; k <= 預算; k++){
            //主要是我們假設卡車容量有1 G，
            //總預算有1 n
            //我們透過紀錄，在卡車容量是G-1 的情況，卡車現在預算- 這種水果預算時，
            //有沒有比現在的dp[卡車容量][預算] 來得小，有就替換
            dp[j][k] = min(dp[j][k], dp[j-1][cost - 水果買入價] + 水果賣出價 )
        }
    }
}

//想要找到最高的預算就是
cout << dp卡車容量預算[][];
```

5.2 LCS

```
#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define N 120
```
```

```cpp
using namespace std;
int n ;
string strA , strB ;
int t[N*N] , d[N*N] , num[N*N] ; //t and d 是LIS 要用到
// d 用來記住LIS 中此數字的前一個數字
// t 當前LIS 的數列位置
// num 則是我們根據strB 的字元生成數列，用來找出最長LIS 長度
map<char,vector<int>> dict ; //記住每個字串出現的index 位置

int bs(int l , int r , int v ){ //binary search
    int m ;
    while(r>l){
        m = (l+r) /2 ;
        if(num[v] > num[t[m]]) l = m+1 ;
        else if (num[v] < num[t[m]]) r = m ;
        else return m ;
    }
    return r ;
}

int lcs(){
    dict.clear() ; //先將dict 先清空
    for(int i = strA.length()-1 ; i > 0 ; i--) dict[strA[i]].push_back(i) ;
    // 將每個字串的位置紀錄並放入vector 中，請記住i = strA.length() -1 才可以達到逆續效果

    int k = 0 ; //紀錄生成數列的長度的最長長度
    for(int i = 1 ; i < strB.length() ; i++){ // 依據strB 的每個字元來生成數列
        for(int j = 0 ; j < dict[strB[i]].size() ; j++)
        //將此字元在strA 出現的位置放入數列
            num[++k] = dict[strB[i]][j] ;
    }
    if(k==0) return 0 ; //如果k = 0 就表示他們沒有共同字元都沒有於是就直接輸出0

    d[1] = -1 , t[1] = 1 ; //LIS init
    int len = 1, cur ; // len 由於前面已經把LCS = 0 的機會排除，於是這裡則從1 開始

    // 標準的LIS 作法，不斷嘗試將LCS 生長
    for(int i = 1 ; i <= k ; i++ ){
        if(num[i] > num[t[len]]) t[++len] = i , d[i] = t[len-1] ;
        else{
            cur = bs(1,len,i);
            t[cur] = i ;
            d[i] = t[cur-1];
        }

        //debug
// for(int i = 1 ; i <= k ; i++)
// cout << num[t[i]] << ' ' ;
// cout << '
      n' ;
    }
    return len ;

}
```

5.3 LIS

```cpp
const int N = 100;
int s[N];       // sequence
int length[N];  // 第x 格的值為s[0...x] 的LIS 長度

int LIS()
{
    // 初始化。每一個數字本身就是長度為一的LIS。
    for (int i=0; i<N; i++) length[i] = 1;

    for (int i=0; i<N; i++)
        // 找出s[i] 能接在哪些數字後面，
        // 若是可以接，長度就增加。
        for (int j=0; j<i; j++)
            if (s[j] < s[i])
                length[i] = max(length[i], length[j] + 1);

    // length[] 之中最大的值即為LIS 的長度。
    int l = 0;
    for (int i=0; i<N; i++)
        l = max(l, length[i]);
    return l;
}
```

5.4 Directed Acyclic Graph

是一種沒有、有向圖，因此

circle DAG 不走回頭路、不斷向前進，永遠都從起點通往到對岸的另外一邊，讓所有點都可以碰觸到終點。但你遇到

DAG 的題目時，你可以使用以下方式解決

DP
* 在起點超過一個以上時使用

SPFA
* 在起點只有一個時使用

6 大衛-圖論

6.1 歐拉回路

```cpp
// Euler Circuit 歐拉迴路介紹
// 每一個邊都只經過一次的前提下，可以從某一個點開始出發，順利經過每一個點
// 分成無向圖、有向圖進行討論

// 定義名詞
// * 入邊，從其他的點進來
// * 出邊，出去至其他的點
// 無向圖
// * 每個邊都是偶數條邊，且會相互連通
// * 且一條邊中，入邊或出邊位置可任意對調
// * 因此我們可以得知，只要無向圖存在Euler Circuit 歐拉迴路
// - 如果起點與終點相同，則沒有一個點的邊數是奇數，有進就有出，因此一定有兩個邊
// - 如果起點與終點不同，則起點與終點的邊數則是奇數，因為一開始出發點沒有入邊，終點則沒有出邊，其他的點則都必須有兩個邊


int n, kase = 0, a, b;
vector<int> edge[MAXN]; // 迅速得知邊長
int g[MAXN][MAXN]; // 判斷這個邊有沒有被用過
int degree[MAXN]; //計算邊數
vector<pair<int,int> > record;

void euler(int root){

    for(int it: edge[root]){
        if(!g[root][it]) continue;
        g[root][it]--; //這個邊被使用過
        g[it][root]--; //這個邊被使用過
        euler(it);
        cout << "root it " << root << " " << it << "\n";
        record.push_back({root, it}); //記得逆序，因為遞迴，會將後面的dfs 先print 出來
    }
}


int main(){
    cin >> n;
    for(int i = 0; i < n; i++){
        cin >> a >> b;
        edge[a].push_back(b); //加入邊
        edge[b].push_back(a);
        g[a][b]++; //這個邊沒被使用過
        g[b][a]++;
        degree[a]++; //這個點新增一個邊
        degree[b]++;
    }

    int flag = 0;
    for(int i = 0; i < n; i++){ //判斷有幾個點的邊為奇數
        if(degree[i] % 2 != 0){
            flag++;
        }
    }

    if(!(flag == 0 || flag == 2)) cout << "can't find euler path\n";
    else{
        record.clear(); //不斷遞迴，找出歐拉路徑
        euler(a);
        for(auto it: record) cout << it.first << " " << it.second << '\n';
    }
    //cout << "";
}
```

6.2 floyd 最短路徑

```
// 能夠針對有、無權重的有向圖做出全點全源最短路徑演算法。
// 全點全源：任意點到任意點的最短距離

// 時間複雜度為O(n^3)，n 為頂點

#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define MAXN 120
#define int long long
#define INF 0x3f3f3f3f
using namespace std;
int t, n, r;
int u, v, c;
int start, destination, kase = 1;
int dist[MAXN][MAXN];

void floyd(){
    for(int k = 0; k < n; k++){ //以k 為中繼點
        for(int i = 0; i < n; i++){ //從i 出發
            for(int j = 0; j < n; j++){ //抵達j
                //如果i to j，經過k 會比較快就替換答案
                if(dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

}

void print(){ //印出最短距離圖
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            printf("%10d ", dist[i][j]);
        }
        printf("\n");
    }
}

int32_t main()
{
#ifdef LOCAL
    freopen("in1.txt", "r", stdin);
#endif // LOCAL
    cin >> t;
    while(t--){
        cin >> n >> r;
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                dist[i][j] = INF; //一開始i 都無法抵達j 節點
            }
            dist[i][i] = 0; //但是自己可以抵達自己
        }
        for(int i = 0; i < r; i++){
            cin >> u >> v;
            dist[u][v] = 1; //加入邊
            dist[v][u] = 1; //考慮雙向邊
        }

        floyd();
        int ans = 0;
        cin >> start >> destination;
        cout << dist[start][destination] << "\n" //輸出起點到終點的最短距離
        //print();

    }
    return 0;
}
```

6.3 最小生成樹

```
# Kruskal

## 用途找出無向圖的最小生成樹


## 時間複雜度
O(ElogE)

## 核心概念並查集、
Greedy

## Code
```c=
#include <bits/stdc++.h>
#define INF 0x3f3f3f3f
#define MAXN 105
using namespace std;

struct edge{
    int u,v,c;
};
bool cmp(edge a, edge b){
    return a.c<b.c;
}

vector<edge> Edge;
vector<edge> MST;
int parent[MAXN];
int n,m;

void init(){
    for(int i=0;i<MAXN;i++) parent[i]=-1;
}

int find_root(int id){
    if(parent[id]==-1) return id;
    return parent[id]=find_root(parent[id]);
}

void merge(int a,int b){
    int root_a=find_root(a),root_b=find_root(b);
    parent[root_b]=root_a;
}

void kruskal(){
    sort(Edge.begin(),Edge.end(),cmp);
    for(auto& i:Edge){
        int root_u=find_root(i.u),root_v=find_root(i.v);
        if(root_u!=root_v){
            MST.push_back(i);
            merge(i.u,i.v);
        }
    }
}

int main(void){
    init();
    cin>>n>>m;
    int a,b,c;
    while(m--){
        edge temp;
        cin>>a>>b>>c;
        Edge.push_back({a,b,c});
    }
    kruskal();
    for(auto& i:MST) cout<<i.u<<' '<<i.v<<' '<<i.c<<'\n';
}
```
```

6.4 找圖中的橋find bridge

```
// 四個陣列，一個vector edge 紀錄題目的邊
// depth 記錄當前深度
// low 紀錄當前節點，能返回的最淺深度是多少
// visit 紀錄是否有走訪過
// ancestor 為disjoint set，將所有橋的節點放在一起

#define MAXN
vector<int> edge[MAXN];
int visit[MAXN], depth[MAXN], low[MAXN];
int ancestor[MAXN];
int cnt = 1

int find_root(int x){
    if(ancestor[x] != x) return ancestor[x] = find_root(ancestor[x]);
    return ancestor[x];
}

void find_bridge(int root, int past){ //找到橋點
    visit[root] = 1; //表示走訪過
    depth[root] = low[root] = cnt++; //邏輯證明2.1
```

```
    for(int node: edge[root]){ //不斷遍地
        //因為是無向邊，因此雙向同個edge 不是bridge
        if(node == past) continue;
        if(visit[node]) low[root] = min(low[root], depth[node]); //邏輯證明2.2
        else{
            //先進行DFS，往下找其他的node 有沒有辦法回到曾經走放過的節點
            find_bridge(node, root);
            low[root] = min(low[root], low[node]); //邏輯證明2.3
            if(low[node] > depth[root]){  //邏輯證明2.4
                int fa_node = find_root(node); //進行disjoint merge
                int fa_root = find_root(root);
                //cout << "fa " << fa_node << " " << fa_root << "
                      n";
                ancestor[fa_node] = fa_root;
            }
        }
    }
}
```

6.5 拓樸排序

```
#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define MAXN 120
using namespace std;
int n, m, a, b;
int cnt[MAXN]; //記錄關係，以此節點為後面，而有多少節點在其前面
vector<int> root[MAXN], ans;
//root 記錄關係，以此節點為前面，而另一節點就在後面(vector.push_back)
//ans 答案序列，拓樸排序的序列

void topo(){
    for(int i = 0; i < m; i++){ //不斷輸入
        cin >> a >> b; //輸入記錄關係，a 是前者b 是後者
        root[a].push_back(b);  //記錄關係，記錄a 有多少後面節點，並且記錄。
        cnt[b]++; //記錄有幾個前面節點，如果b 是後面關係時。
    }

    deque<int> q; //用來判斷有哪些節點現在已經可以直接被放到答案序列
    for(int i = 1; i <= n; i++){
        if(cnt[i] == 0) q.push_back(i);
        //在記錄關係中，如果以此節點為後面，沒有節點在前面就加入q
    }

    int now; //暫存bfs(q) 當前的節點
    ans.clear(); //答案序列清空
    while(ans.size() != n){ //如果答案序列的長度跟題目給的長度一樣就跳出
        if(q.empty()) break; //如果沒有節點可以直接被放入答案序列就跳出
        now = q.front(); q.pop_front(); //把當前節點給now
        ans.push_back(now); //將now 放入答案序列
        for(auto it: root[now]){ //由於now 節點被放入答案陣列，
        //之前的記錄關係就不須記錄，因為放到答案陣列就剩下的後面節點就必定在後面

            cnt[it]--; //將所有原本在記錄關係中後面的節點-1，減少了一個記錄關係
            if(cnt[it] == 0) q.push_back(it); //如果都沒有記錄關係就可以放到q
        }
    }
    if(ans.size() == n){ //如果答案序列跟n 一樣，表示可以成功排出拓樸排序，就輸出答案
        for(int i = 0; i < ans.size(); i++) cout << ' ' << ans[i];
        cout << '\n';
    }
}
```

6.6 Component Kosaraju's Algorithm 找出SCC

```
// # Kosaraju's Algorithm 介紹
// Kosaraju's Algorithm 可以找出有向圖的SCC

// Sridge-connected Component(強連通分量)
// Bridge-connected Component 所有兩點之間雙向皆有路可以抵達

// ## Kosaraju's Algorithm 原理
// ### 證明
// /* 如果是A、B、C 三個點都為SCC，那我從A 反方向走或正方向走都能走到A
// /* 其中圖中有一條邊為A -> D
// - 如果有一個邊是D -> A，那我們就可以表示A、B、C、D 都是SCC
// /* 因此我們準備一張反向圖，一樣從A 出發
// - 題目給的圖如果是A -> B，則反向圖為B -> A
// /* 走訪A、B、C
// - 同理，如果我能夠滿足此上述條件就表示A、B、C 為SCC
// /* 無法走訪A、D
// /* 不可以的話，他們就不是同一組SCC


#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define MAXN 1020
#define int long long
using namespace std;
vector<int> edge[MAXN]; //題目圖
vector<int> rev_edge[MAXN]; //反向圖
vector<int> path; //紀錄離開DFS 的節點順序
int visit[MAXN];
int group[MAXN]; //判斷此節點在哪一個組
int cnt, a, b, q;

void dfs1(int root){
    if(visit[root]) return;
    visit[root] = 1;

    for(auto it: edge[root]){
        dfs1(it);
    }
    path.push_back(root); //紀錄DFS 離開的節點
}

void dfs2(int root, int ancestor){
    if(visit[root]) return ;

    visit[root] = 1;
    group[root] = ancestor; //root 跟ancestor 在同一個SCC
    for(auto it: rev_edge[root]){
       dfs2(it, ancestor);
    }
}

void kosoraju(){
    for(int i = 0; i < q; i++){ //q 為邊的長度
        cin >> a >> b;
        edge[a].push_back(b); //題目圖
        rev_edge[b].push_back(a); //反向圖
    }

    memset(visit, 0, sizeof(visit));
    path.clear();
    for(int i = 1; i < cnt; i++){ //第一次DFS
        if(!visit[i]) dfs1(i);
    }

    memset(visit, 0, sizeof(visit));
    memset(group, 0, sizeof(group));
    for(int i = path.size()-1; i >= 0; i--){ //尋找以path[i] 為主的SCC 有哪些節點
        if(!visit[path[i]]){
            dfs2(path[i], path[i]);
        }
    }

}
```

6.7 dijkstra

```
// # Dijkstra'a Algorithm 介紹
// 能夠針對有權重的有向圖做出單點全源最短路徑演算法。
// 時間複雜度為O((E+V)logV)，E 為邊、V 為頂點
// * Dijkstra 變化題，可以擴增dist
// 如，dist[node][第n短路徑]、dist[node][奇偶數路徑]、可以走重複路徑時，則使用visit[i]，來避免deque 裡面有相同節點

#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define int long long
#define MAXN 10020
#define INF 2100000000
using namespace std;
struct Edge{ // 我們使用Edge struct 實做(root, cost)
```

```
    int v, c;
    Edge(): v(0), c(0) {}
    Edge(int _v, int _c): v(_v), c(_c) {}

    bool operator < (const Edge& other) const{
        return c < other.c; //遞減排序，決定priority_queue 的方式
        //return c > other.c; //遞增排序
    }
};

int c, v;
int a, b, cost;
vector<Edge> edge[MAXN]; //放入題目的邊
int dist[MAXN]; //從root 出發到x 邊的最短距離

void dijkstra(int root){
    deque<Edge> q;
    q.push_back({root,0}); //初始放入開始點
    dist[root] = 0; //自己到自己成本為零

    int cost;
    while(!q.empty()){
        Edge node = q.front(); q.pop_front();
        //cout << node.v << " " << node.c << "
            n";
        for(auto it: edge[node.v]){
            cost = dist[node.v] + it.c; //分析3
            if(cost < odd_dist[it.v]){
                q.push_back({it.v, cost});
                odd_dist[it.v] = cost;
            }
        }
    }
}


int32_t main()
{
#ifdef LOCAL
    freopen("in1.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
#endif // LOCAL
    while(cin >> c >> v){
        for(int i = 1; i <= c; i++){ //清除邊、重置距離
            edge[i].clear();
            dist[i] = INF;
        }
        for(int i = 0; i < v; i++){ //加入邊
            cin >> a >> b >> cost;
            edge[a].push_back({b,cost}); //單向時使用
            edge[b].push_back({a,cost}); //雙向時使用
        }
        dijkstra(root); //root 為任移值，為開始的點
        if(dist[x] == INF) cout << "-1\n"; // root -> x 最短距離為多少，無法抵達輸出-1
        else cout << dist[x] << "\n"; //可以抵達則輸出。
    }
    return 0;
}
```

6.8 二分匹配、二分圖

```
// 在介紹最大二分匹配時，必須先介紹二分圖。
// 二分圖是一種圖的特例，二分圖的結構為，X 群體的每一個點都有連結到至少一個以上Y 群體的點、T 群體的每一個點都有連結到至少
      一個以上X 群體的點，且X、Y 群體各自沒有邊互相連接。
// 二分匹配就是，每一個x 節點只能連到一個y 個節點、每一個y 節點只能連到一個x 個節點，舊式二分匹配，類似於一夫一妻制。
// 我們使用Augmenting Path Algorithm 實作，時間複雜度為O(VE)，V 是頂點、e 是邊

//二分匹配變化，如果遇到棋盤，其中以座標(x,y) 為例，則x,y 軸只能有此點，那也是二分圖，這邊則是將x 軸與y 軸配對。


vector<int> edge[MAXN];
int mx[MAXN], my[MAXN], vy[MAXN]; //matchX, matchY, visitY

bool dfs(int x){
    for(auto y: edge[x]){ //對x 可以碰到的邊進行檢查
        if(vy[y] == 1) continue; //避免遞迴error
        vy[y] = 1;
        if(my[y] == -1 || dfs(my[y])){ //分析3
            mx[x] = y;
            my[y] = x;
            return true;
        }
    }
    return false; //分析4
}

int bipartite_matching(){
    memset(mx, -1, sizeof(mx)); //分析1,2
    memset(my, -1, sizeof(my));
    int ans = 0;

    for(int i = 1; i <= cnt; i++){  //對每一個x 節點進行DFS(最大匹配)
        memset(vy, 0, sizeof(vy));
        if(dfs(i)) ans++;

    }
    return ans;
}
```

7 大衛-資料結構

7.1 並查集

```
#define MAXN 2000
void init(){
    for(int i = 0; i < MAXN; i++){
        tree[i] = i;
        cnt[i] = 1; //cnt 為數量，也就是每一個集合的數量，一開始都是1，因為只有自己。
    }
}

int find_root(int i){
    if(tree[i] != i) //如果tree[i] 本身並不是集合中的代表元素，
    //表示這個集合中有其他元素，並且其他元素才是代表元素
        return tree[i] = find_root(tree[i]); //遞迴，將tree[i] 的元素在進行查詢，
        //並將代表元素設為現在的tree[i]
    return tree[i]; //回傳代表元素
}

void merge(int a, int b){
    rx = find_root(tree[a]); //找出find_root(tree[a]) 的代表元素
    ry = find_root(tree[b]); //找出find_root(tree[b]) 的代表元素
    if(rx != ry) //如果不一樣就合併
        tree[ry] = rx; //要合併的是代表元素，不是tree[b]
        cnt[rx] += cnt[ry]; //將原本另一集合的數量加到這集合，因為他們合併了
        cnt[ry] = 0;  //由於合併，因此將原本獨立的集合數量歸0
}
```

7.2 線段樹

```
#define INF 0x3f3f3f
#define Lson(x) (x << 1)
#define Rson(x) ((x << 1) +1)
#define MAXN 題目陣列最大長度

struct Node{
    int left; // 左邊邊界
    int right; //右邊邊界
    int value; //儲存的值
    int z; //區間修改用，如果沒有區間修改就不需要
}node[4 * N ];

void question(){
    for(int i = 1; i <= 10; i++) num[i] = i * 123 % 5;
    // num 為題目產生的一段數列
    // hash 函數，讓num 的i 被隨機打亂
}

void build(int left , int right , int x = 1 ){
    // left 為題目最大左邊界，right 為題目最大右邊界，圖片最上面的root 為第一個節點
    node[x].left = left ; //給x 節點左右邊界
    node[x].right = right ;
    if(left == right){ //如果左右邊界節點相同，表示這裡是葉節點
        node[x].value = num[left] ;  //把num 值給node[x]
        //這裡的num 值表示，我們要在value 要放的值
```

```
        return ; //向前返回
    }
    int mid = (left + right ) / 2 ; //切半，產生二元樹

    //debug
    //cout << mid << '
         n' ;
    //cout << x << ' ' << node[x].left << ' ' << node[x].right << ' ' << '
         n' ;

    build(left , mid , Lson(x)) ; //將區間改為[left, mid] 然後帶給左子樹
    build(mid + 1 , right , Rson(x)) ; //將區間改為[mid+1, right] 然後帶給右子樹
    node[x].value = min(node[Lson(x)].value  , node[Rson(x)].value ) ;
    //查詢左右子樹哪個數值最小，並讓左右子樹最小值表示此區間最小數值。
}

void modify(int position , int value , int x = 1 ){ //修改數字
    if(node[x].left == position && node[x].right == position ){ //找到葉節點
       node[x].value = value ; //修改
       return ; //傳回
    }
    int mid = (node[x].left + node[x].right ) / 2 ; //切半，向下修改
    if(position <= mid ) //如果要修改的點在左邊，就往左下角追蹤
        modify(position , value , Lson(x) );
    if(mid < position ) //如果要修改的點在右邊，就往右下角追蹤
        modify(position , value , Rson(x)) ;
    node[x].value = min(node[Lson(x)].value , node[Rson(x)].value );
    //比較左右子樹哪個值比較小，較小值為此節點的value

}

#define INF 0x3f3f3f

int query(int left , int right , int x = 1 ){
    if(node[x].left >= left && node[x].right <= right)
        return node[x].Min_Value ;
    //如果我們要查詢的區間比當前節點的區間大，那我們不需再向下查詢直接輸出此答案就好。
    // 例如我們要查詢[2, 8]，我們只需要查詢[3, 4]，不須查詢[3, 3]、[4, 4]，
    // [3, 4] 已經做到最小值查詢

    push_down(x);//有區間修改時才需要寫
    int mid = (node[x].left + node[x].right ) / 2 ; //切半，向下修改
    int ans = INF ; //一開始先假設答案為最大值

    if( left <= mid ) //如果切半後，我們要查詢的區間有在左子樹就向下查詢
        ans = min(ans , query(left , right , Lson(x))) ; //更新答案，比較誰比較小
    if(mid < right ) //如果切半後，我們要查詢的區間有在右子樹就向下查詢
        ans = min(ans , query(left , right , Rson(x))) ; //更新答案，比較誰比較小
    return ans ; //回傳答案
}
```

8 大衛-字串

8.1 KMP

```
// 給你一字串，請新增字元讓這字串變成迴文，但新增字元數量要最少。
// 迴文：從左邊讀與從右邊讀意思都一樣
// 題目善意提示：這題不要給經驗不足的新手做M
// KMP algorithm 介紹
// 在線性時間內找出段落(Pattern) 在文字(text) 中哪裡出現過。
// 對Pattern 找出次長相同前綴後綴，在使用DP 將時間複雜度壓縮

string  strB ;
int b[MAXN] ;
// b[] value 表示strB當下此字元上次前綴的index，如果已經沒有前綴則設定-1

void kmp_process(){
    int n = strB.length() ,i = 0 ,  j = -1 ;
    // j = 前綴的長度
    //strB 是pattern , j = -1 時代表沒有辦法再回推到前一個次長相同前綴
    b[0] = -1 ;
    // 由於strB[0] 絕對沒有前綴所以設定-1
    while(i < n ){ //對從Pattern 的第0 個字元到第i 字元找出次長相同前綴
        while(j >= 0 && strB[i] != strB[j]) j = b[j] ;
        // j >= 0 代表還可以有機會找出次長相同前綴
        // strB[i] != strB[j] 則代表他們字元不同，於是在這裡把j 值設為b[j]
        // 當j 只要被設定成-1 就代表完全沒有次長相同前綴
        i++ ; j++ ;
        b[i] = j ;
        // strB[i] 上次前綴的index 值或是將j 設定成0 而不設定成-1 是因為
        // 他有可能會是strB[0] 長度只有1 的前綴
    }

    //debug 供應測試用
    // for(int k = 0 ; k <= n ; k++)
    // cout << b[k] << ' ' ;
    // cout << '
         n' ;
}

string strA ;
//strA 是text

void kmp_search(){
    int n = strA.length() , m=strB.length() , i=0 , j=0  ;
    while(i < n ){ //對從text 找出搜尋哪裡符合Pattern
        while(j >= 0 && strA[i] != strB[j]) j = b[j] ;
        // j >= 0 代表還可以有機會是pattern 的前綴
        // strA[i] != strB[j] 則代表他們字元不同，於是在這裡把j 改為b[j]
        // b[j] 說明請看kmp_process 宣告b[j] 時的解釋
        i++ ; j++ ;
        if (j == m) { // j 已經跟pattern 的長度相同了
            printf("P is found at index %d in T\n", i - j);
            // 告訴使用者在哪裡找出
            j = b[j];
            // 將j 設定成此字元上次前綴的index
        }
    }
}
```

8.2 最短修改距離

```
// Minimum Edit Distance 介紹
// 可以透過刪除、插入、替換字元來達到將A 字串轉換到B 字串，並且是最少編輯次數。
// 此演算法的時間複雜度O(n2)

// 最短修改距離Minimum Edit Distance 應用
// DNA 分析
// 拼寫檢查
// 語音辨識
// 抄襲偵測

int dis[MAXN][MAXN];
//dis[A][B] 指在strA 長度0 to A 與strB 長度0 to B 的最短修改距離為多少
//這裡假設由A 轉換B
string strA , strB ;
int n , m ;
n=strA.length() ;
m=strB.length() ;

int med(){ //Minimum Edit Distance
    for(int i = 0 ; i <= n ; i++) dis[i][0] = i ;
    // 由於B 是0 ，所以A 轉換成B 時每個字元都要被進行刪除的動作
    for(int j = 0 ; j <= m ; j++) dis[0][j] = j ;
    // 由於A 是0 ，所以A 轉換成B 時每個字元都需要進行插入的動作
    for(int i = 1 ; i <= n ; i++){ // 對strA 每個字元掃描
        for(int j = 1 ; j <= m ; j++){ // 對strB 每個字元進行掃描
            if(strA[i-1] == strB[j-1]) dis[i][j] = dis[i-1][j-1] ;
            // 如果他們字元相同則代表不需要修改，因此修改距離直接延續先前
            else dis[i][j] = min(dis[i-1][j-1], min(dis[i-1][j] , dis[i][j-1]))+1;
            // 因為她們字元不相同，所以要詢問replace , delete , insert 哪一個編輯距離
            // 最小，就選擇他+1 來成為目前的最少編輯距離
        }
    }
}

return dis[n][m] ; // 這就是最少編輯距離的答案

// QUESTION: 現在的我們知道最少編輯距離的答案，那我們可以回推有哪些字元被編輯嗎？
// 那當然是可以的阿XD，只是寫起來比較麻煩。通常這種答案會有很多種，依照題目的要求通常只需要你輸出一種方式即可。除非是毒瘤

// 實現方式如下：
// 由於這回推其實也就只是一個簡單的遞迴你能夠推得出DP 就可以知道要怎麼回推哪些字元被編輯，於是我就在程式碼上旁寫下說明來
    幫助讀者閱讀。希望能夠幫助到
```