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()
```

```python
# ## 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 官-數學

2.1 找因數

```cpp
int num = 某數;
for(int i = 1; i <= (int)sqrt(num); i++)
{
    if(num % i == 0)
    {
        // 得到因數i
        // 4 = 2 * 2，只保留一個2
        if(num / i != i)
        {
            // 12 = 3 * 4，像這樣，只要找到3，就知道4
            // 得到因數num / i
        }
    }
}
```

2.2 公因數

// * $GCD(a_1+b_j, \ldots, a_n+b_j) = GCD(a[0]+b[j], a[1]-a[0], a[2]-a[1], \ldots\ldots, a[n-2]-a[n-3], a[n-1]-a[n-2])$
// $a[0] < a[1] < a[2] < a[3] \ldots$
// 用法
// * 因為只有$b\ j$ 是不固定的，所以求$GCD(a_1 + b_j, \ldots, a_n + b_j)$ 就只要算$GCD(a[0] + b[j], baseGcd)$
// * $baseGcd = GCD(a[1] - a[0], a[2] - a[1], \ldots\ldots, a[n - 2] - a[n - 3], a[n - 1] - a[n - 2])$

```cpp
#include <iostream>
#include <map>
#include <queue>
#include<bits/stdc++.h>
using namespace std;
const int maxn = 2e5 + 10;
const int mod=1e9+7;
typedef long long ll;
map<string, int>mp;
int gcd(int a,int b){
        return b==0?a:gcd(b,a%b);
}
ll a[maxn];
int main() {
        int n,m;cin>>n>>m;
        for(int i=0;i<n;i++) cin>>a[i];
        sort(a,a+n);
        ll g=0;
        for(int i=1;i<n;i++) g=__gcd(a[i]-a[i-1],g);
        for(int i=0;i<m;i++){
                ll x;cin>>x;
                cout<<__gcd(x+a[0],g)<<" ";
        }
        return 0;
}
```

2.3 求導數

// * 輸入$f(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$
// * 求導數$f'(x) = a_0nx^{n-1} + a_1(n - 1)x^{n-2} + \ldots + a_{n-1}$
// * 將公式重組，可以省略多餘的次方運算
// * 如此反覆提取公因數x ，最後將函數化為$f(x) = (((a_nx + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$

2.4 Sum of Product

```
n = 3
SOP(Pk) = {0, 1, 2} 內部可調換()

P_k Permutation SOP(P_k)
P1    0 1 2        2
P2    0 2 1        2
P3    1 0 2        2
P4    1 2 0        2
P5    2 0 1        2
P6    2 1 0        2輸出有幾種

 SOP(Pk)

* 網路上有公式和序列
    * 1, 1, 1, 3, 8, 21, 43, 69, 102, 145, 197, 261, 336, 425, 527, 645, 778, 929, 1097, 1285, 1492,
          1721, 1971, 2245, 2542, 2865, 3213, 3589, 3992, 4425, 4887, 5381, 5906, 6465, 7057, 7685,
          8348, 9049, 9787, 10565, 11382, 12241, 13141, 14085, 15072, 16105
    * For n >= 7
        * a(n) = (n^3-16*n+27)/6 (n is odd)
        * a(n) = (n^3-16*n+30)/6 (n is even)
```

2.5 法里數列

```
* 求 Farey sequences(Fn) 的第 k 個分數
* 利用遞推式不斷求出新的數值
    * 遞推式 :
        * num = (n + b1)(b2)
        a3 = num * a2 - a1;
        b3 = num * b2 - b1;
```

3 立委-幾何

3.1 Center of Masses (Polygon)

```cpp
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>

using namespace std;

// 求重心
```

```cpp
// 把多邊形切開成許多個三角形，分別計算各個三角形的重心。然後以三角形面積作為權重，計算三角形重心的加權平均值，就得到多邊
    形的重心

class Point {
   private:
   public:
    double x, y;
    Point() : x(0), y(0) {}
    Point(double X, double Y) : x(X), y(Y) {}
    ~Point() {}

    bool operator<(Point const& r) const {
        return x < r.x || (x == r.x && y < r.y);
    }

    bool operator==(Point const& r) const {
        return x == r.x && y == r.y;
    }

    Point& operator+(Point const& r) const {
        return *(new Point(x + r.x, y + r.y));
    }

    Point& operator-(Point const& r) const {
        return *(new Point(x - r.x, y - r.y));
    }

    double cross(Point const& r) const {
        return x * r.y - y * r.x;
    }
};

Point massCenter(vector<Point> polygon) {
    if (polygon.size() == 1) {
        return polygon[0];
    } else if (polygon.size() == 2) {
        return Point((polygon[0].x + polygon[1].x) / 2, (polygon[0].y + polygon[1].y));
    }
    double cx = 0, cy = 0, w = 0;
    for (int i = polygon.size() - 1, j = 0; j < polygon.size(); i = j++) {
        double a = polygon[i].cross(polygon[j]);
        cx += (polygon[i].x + polygon[j].x) * a;
        cy += (polygon[i].y + polygon[j].y) * a;
        w += a;
    }

    return Point(cx / 3 / w, cy / 3 / w);
}
```

3.2 convexHull

```cpp
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>

using namespace std;

// 首先升序排序所有按x升序（x相同按y升序），除重复后得到序列p1,p2...，然后p1和p2放到凸包中。p3始，新在凸包\前"方向的左
    （叉正），否（叉或0）依次除最近加入到凸包的，直到新在左。
// 左到右做一次之后得到的是\下凸包"，然后右向左做一次得到\上凸包"，合起就是完整的凸包。复度O(nlogn)。

class Point {
   private:
    double x, y;

   public:
    Point() : x(0), y(0) {}
    Point(double X, double Y) : x(X), y(Y){};
    ~Point(){};

    bool operator<(Point const& r) {
        return x < r.x || (x == r.x && y < r.y);
    }

    bool operator==(Point const& r) {
        return x == r.x && y == r.y;
    }

    Point& operator+(Point const& r) const {
        return *(new Point(x + r.x, y + r.y));
    }

    Point& operator-(Point const& r) const {
        return *(new Point(x - r.x, y - r.y));
    }

    double cross(Point const& r) const {
        return x * r.y - y * r.x;
    }

    Point& rotate(double degree) const {
        return *(new Point(x * cos(degree) - y * sin(degree), x * sin(degree) + y * cos(degree)));
    }
};

vector<Point> convexHull(vector<Point>::iterator first, vector<Point>::iterator last) {
    vector<Point> p(first, last);
    sort(p.begin(), p.end());
    p.resize(unique(p.begin(), p.end()) - p.begin());
    if (p.size() < 3) return p;

    vector<Point> result;
    for (int i = 0; i < p.size(); i++) {
        while (result.size() >= 2 && (result.back() - result[result.size() - 2]).cross(p[i] - result.
              back()) <= 0) result.pop_back();
        result.push_back(p[i]);
    }

    int k = result.size();
    for (int i = p.size() - 2; i >= 0; i--) {
        while (result.size() >= k + 1 && (result.back() - result[result.size() - 2]).cross(p[i] -
              result.back()) <= 0) result.pop_back();
        result.push_back(p[i]);
    }
    result.pop_back();
    return result;
}

double area(vector<Point>::iterator first, vector<Point>::iterator last) {
    double ans = 0;
    for (auto i = first + 1; i + 1 < last; i++) {
        ans += (*i - *first).cross(*(i + 1) - *first);
    }
    return ans / 2;
}
```

3.3 Intersection(Line and Line)

```cpp
#include <iomanip>
#include <iostream>
using namespace std;

// 斜率一樣代表平行或重合
// m = dy/dx
// m1 = m2
//      dy1/dx1 = dy2/dx2
//      dy1 * dx2 = dy2 * dx1

// 求兩直線交點
// 利用正弦定理可知
// a/sin a = b/sin b = c/sin c = 2R
//      a = c*sina/sinc
// 又T = |u×b|/|a×b| = |u| sin θ/|a| sin β  (sin θ = sin a)
// 故T * a⃗ = EB⃗

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 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;
}
```

3.4 Intersection(Line and Point)

```
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;

// 確定點在線的上面或下面假設有一直線通過點P ，且方向向量為v，求某一點Q 在線的上部分或下部分或點上
// 公式：→
// PQ × v
// 結果：
// 負數->上部分
// 正數->下部分
// 0 -> 線上
// 可利用此公式當跨立實驗的判斷基準

// 快速排斥實驗
// 若兩線段四方形區域未重疊則此兩線段必不相交
// 求P1P2 及Q1Q2
// 設Q1(q1x, q1y) ....  以此類推
// - 程式碼：
// min(p1x,p2x) <= max(q1x,q2x) && min(q1x,q2x) <= max(p1x,p2x) && min(p1y,p2y) <= max(q1y,q2y) &&
    min(q1y,q2y) <= max(p1y,p2y)

// 跨立實驗
// 若有AB 及CD
// A 與B 在CD 兩側且C 與D 在AB 兩側，則通過跨立實驗
// 通過跨立實驗不代表兩線相交，需要同時通過快速排斥實驗才是兩線段相交

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;
}
```

3.5 Pack Polygon Circle

```
#include <cmath>
#include <iostream>

using namespace std;

// 找三角形外心
// (x1-x)*(x1-x)-(y1-y)*(y1-y)=(x2-x)*(x2-x)+(y2-y)*(y2-y);
// (x2-x)*(x2-x)+(y2-y)*(y2-y)=(x3-x)*(x3-x)+(y3-y)*(y3-y);

// 化簡得到：
// 2*(x2-x1)*x+2*(y2-y1)y=x2^2+y2^2-x1^2-y1^2;
// 2*(x3-x2)*x+2*(y3-y2)y=x3^2+y3^2-x2^2-y2^2;
// 令A1=2*(x2-x1)；
```

```cpp
// B1=2*(y2-y1);
// C1=x2^2+y2^2-x1^2-y1^2;
// A2=2*(x3-x2);
// B2=2*(y3-y2);
// C2=x3^2+y3^2-x2^2-y2^2;
// 即
// A1*x+B1y=C1;
// A2*x+B2y=C2;

// 最後根據克拉默法則
// x=((C1*B2)-(C2*B1))/((A1*B2)-(A2*B1));
// y=((A1*C2)-(A2*C1))/((A1*B2)-(A2*B1));

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;
}
```

3.6 Two Point and Circle

```cpp
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;
// - 餘弦定理
// - 設三角形三邊A、B、C 與三對角θa、θb、θc
// - A^2 = B^2 + C^2 - 2BC cos θa
// - cos θa = (B^2+C^2-A^2) / 2BC

class Vector {
   public:
    double x, y;
    Vector() : x(0), y(0) {}
    Vector(double X, double Y) : x(X), y(Y) {}
    Vector operator-(const Vector& other_point) const {
        return *(new Vector(x - other_point.x, y - other_point.y));
    }
    double dot(const Vector& other_point) const {
        return x * other_point.x + y * other_point.y;
    }

    double cross(const Vector& other_point) const {
        return x * other_point.y - y * other_point.x;
    }

    double square() const {
        return (x * x + y * y);
    }

    double distance() const {
        return sqrt(square());
    }
};

double p2l_dist(Vector A, Vector B, Vector O) {
    Vector AB = B - A;
    Vector dv(-AB.y, AB.x);
    if (A.cross(dv) * B.cross(dv) >= 0) {
        return min(A.distance(), B.distance());
    }

    Vector AO = O - A;
    return sqrt(AO.square() - pow(AO.dot(AB) / AB.distance(), 2));
}

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

    int n;
    cin >> n;

    while (n--) {
        Vector A, B, O(0, 0);
        double r, ans;
        cin >> A.x >> A.y >> B.x >> B.y >> r;

        if (p2l_dist(A, B, O) + 1e-7 >= r) {
            ans = (A - B).distance();
        } else {
            double AO = A.distance();
            double BO = B.distance();
            double AB = (A - B).distance();
            double a3 = acos((AO * AO + BO * BO - AB * AB) / (2 * AO * BO)) - acos(r / AO) - acos(r /
                BO);
            ans = a3 * r + sqrt(AO * AO - r * r) + sqrt(BO * BO - r * r);
        }
        cout << fixed << setprecision(3) << ans << endl;
    }

    return 0;
}
```

4 立委-圖論

4.1 Kruskal

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

#define MAXN 200020

using namespace std;

int p[MAXN];

class Edge {
   private:
   public:
    int start, to, cost;

    Edge() : start(0), to(0), cost(0) {}
    Edge(int start, int to, int cost) : start(start), to(to), cost(cost) {}

    bool operator<(const Edge& other) const {
        return cost < other.cost;
    }
};
```

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

int main() {
    int n, m;
    while (cin >> n >> m, n != 0) {
        vector<Edge> edges;
        for (int i = 0; i < n; i++) p[i] = i;

        for (int i = 0; i < m; i++) {
            int start, to, cost;
            cin >> start >> to >> cost;
            edges.push_back(Edge(start, to, cost));
        }

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

        int save = 0;

        for (auto i : edges) {
            int start_root = find_root(i.start);
            int to_root = find_root(i.to);
            if (start_root != to_root) {
                p[to_root] = start_root;
            } else {
                save += i.cost;
            }
        }

        cout << save << endl;
    }

    return 0;
}
```

4.2 Shortest Path

```
#include <string.h>

#include <iostream>
#include <queue>
#include <vector>

#define MAXN 10010

using namespace std;

vector<int> edges[MAXN];
int import[MAXN];

int shortest_path(int n, int root) {
    bool visit[n + 1];
    int dist[n + 1];
    memset(visit, false, sizeof(visit));
    memset(dist, 0x3f3f3f, sizeof(dist));

    deque<int> q;
    q.push_back(root);
    dist[root] = 0;

    while (!q.empty()) {
        int node = q.front();
        q.pop_front();

        for (auto i : edges[node]) {
            if (dist[i] > dist[node] + 1) {
                dist[i] = dist[node] + 1;
                if (!visit[i]) {
                    visit[i] = true;
                    q.push_back(i);
                }
            }
        }
    }

    // 這邊依照題目決定code
    int result = 0;
    for (int i = 1; i <= n; i++) {
        if (import[i] > 1) result += dist[i];
    }
    return result;
}

void reset(int n) {
    for (int i = 1; i <= n; i++) {
        edges[i].clear();
    }
    memset(import, 0, sizeof(import));
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        int n, s;
        cin >> n >> s;

        reset(n);

        while (s--) {
            int prev, cur;
            cin >> prev;
            while (cin >> cur, cur > 0) {
                edges[prev].push_back(cur);
                edges[cur].push_back(prev);
                import[prev]++;
                prev = cur;
            }
            import[prev]++;
        }

        int ans = 0, min_dist = INT32_MAX;

        for (int i = 1; i <= n; i++) {
            if (import[i] > 1) {
                int dist = shortest_path(n, i);

                if (dist < min_dist) {
                    min_dist = dist;
                    ans = i;
                }
            }
        }

        cout << "Krochanska is in: " << ans << 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
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

```
#include <iostream>
#include <bits/stdc++.h>
#define MAXN 5020
#define LOCAL
using namespace std;

int a[MAXN];
int T, n, len = 0, cur;

int lis(){
    deque<int> b; //用來產生LIS 長度
    b.push_back(a[0]); //先放入一個數值，以避免b.back() 找不到值
    int temp; //紀錄二分搜尋後找到的位置
    for(int i = 1; i < n; i++){
        if(a[i] > b.back()){ //如果現在這個數字大於此數列中最大的數字
            b.push_back(a[i]); //LIS push back
        }
        else{
            temp = upper_bound(b.begin(), b.end(), a[i]) - b.begin();
            //二分搜尋，找到他適合的位置，前面數字比她小或相等，後面數字大
            temp = lower_bound(b.begin(), b.end(), a[i]) - b.begin();
            //大部分使用upper_bound，少用lower_bound
            b.insert(b.begin()+temp , a[i]); //插入數值在此位置
        }
    }
    return b.size(); //輸出最長LIS 長度
}
```

5.4 約瑟夫問題

```
int josephus(int n, int k){
    int s = 0; //一開始的編號
    for(int i = 2; i <= n; i++) s = (s+k) % i; //第i 輪中，他的位置是第s
    return s+1; //如果題目的編號一開始是1，那我們就加一
}
```

6 大衛-圖論

6.1 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 0x3f3f3f
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;
}
```

6.2 最小生成樹

```
// Kruskal Algorithm 介紹
//
// 主要是在一張圖中組合成一顆樹，其中每一條邊都有一個成本，且要求這顆樹的總和成本必須要是最小。
// 時間複雜度為O(ElogE)
// 主要用來找出一張圖中的最小生成樹、最大生成樹

#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define MAXN 200020
#define int long long
using namespace std;
int n, m;
int a, b, c;
int p[MAXN];

struct edge{
    int u, v, c; //u,v 分別為邊的節點，c 是成本

    edge(): u(0), v(0), c(0) {}
    edge(int u, int v, int c): u(u), v(v), c(c) {}
    bool operator < (const edge& other) const{
        return c < other.c;
    }
};
vector<edge> node;
vector<edge> MST; //最小生成樹

int find_root(int x){
    //cout << "find_root " << x << "
        n";
    if(p[x] != x) return p[x] = find_root(p[x]);
    return x;
}

void kruskal(){
    node.clear();
    MST.clear();
    for(int i = 0; i < n; i++) p[i] = i; //init disjoint set

    for(int i = 0; i < m; i++){
        cin >> a >> b >> c; //輸入邊、成本
        node.push_back({a,b,c});
    }
    sort(node.begin(), node.end()); //排序，這邊排序方式為遞增

    for(edge it: node){
        //cout << it.u << " " << it.v << " " << it.c << "
            n";
        //cout << p[3] << " " << p[4] << "
            n";
        int pu = find_root(it.u); //判斷邊的節點們是否都在同個set
        int pv = find_root(it.v);
        if(pu != pv){ //分析3-1
                p[pv] = pu;
                MST.push_back(it); //記錄此edge
        }
    }

    for(edge it: MST){
        cout << it.u << " " << it.v << " " << it.c << "\n"; //輸出所有邊
    }


}
```

6.3 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;
            }
        }
    }
}
```

6.4 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';
    }
}
```

6.5 Component Kosaraju's Algorithm 找出SCC

```cpp
// # 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]);
        }
    }

}
```

6.6 Tarjan's Algorithm 找出ACC

```cpp
// ## Tarjan's Algorithm 無向圖實作與原理
// ### 原理
// /* 使用DFS 實作
// /* 使用堆疊紀錄每一個經過的點
// /* 找出每一個點最高能回到哪一個點
// /* 如果這個點最高能回朔的點還是自己，則表示這個點往下的所有點都會回朔到他，形成一個SCC，因此將堆疊裡的點刪除，直
      到stack.top() 最高能回朔的點還是自己。


#include <iostream>
#include <bits/stdc++.h>
#define LOCAL
#define int long long
#define MAXN 10020
using namespace std;
int cnt;
vector<int> edge[MAXN]; //圖
int stk[MAXN], in_stk[MAXN]; //stk 是堆疊, in-stk 確認此點是否已在堆疊上
int visit[MAXN]; //是否有走訪過
int lead[MAXN], low[MAXN]; //lead 表示此點為哪一個SCC、low 表示此點最高能回到哪一個點
int stk_index; //堆疊的size


void scc(int root){
    if(visit[root]) return;

    visit[root] = low[root] = cnt++; //因為是第一次接觸，先認定root 只能回到root
    stk[++stk_index] = root; // root 加入stack，stk-index+1
    in_stk[root] = 1; //此點加入stack

    for(auto it: edge[root]){ //DFS
        scc(it);
        //如果scc 完成以後，因為root -> it 是一條邊，如果it 可以返回到的點比root 小，
        //那就改變low[root]
        if(in_stk[it]) low[root] = min(low[it], low[root]);
    }

    //如果low[root] 同時也是root，表示這個點是SCC 起點，把這個點以下的stack，都設定為同組的SCC
    if(visit[root] == low[root]){
        int it;
        do{
            it = stk[stk_index--]; //找出stack.top()
            in_stk[it] = 0; //stack.pop()
            lead[it] = root; //it 的SCC 是root 組
        }while(it != root); //只要it != root，表示還沒有找玩
    }
}

void tarjan(){
    memset(visit, 0, sizeof(visit));
    for(int i = 1; i <= n; i++){
        if(visit[i]) continue;
        stk_index = -1; cnt = 1;
        scc(i);
    }
}
```

6.7 dijkstra

```cpp
// # 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
    }
};
```

```cpp
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

```cpp
// 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

```cpp
#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

```cpp
#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

}

void push_down(int x, int add){ //將懶人標記往下推，讓下一層子樹進行區間修改
    int lson = Lson(x), rson = Rson(x);
    node[lson].z += add;  //給予懶人標記，表示子樹如果要給子樹的子樹區間修改時，
    node[rson].z += add;  //數值要是多少，左右子樹都需要做

    node[lson].v += add; //更新左右子樹的值
    node[rson].v += add;
}

void update(int a, int b, int cmd, int x = 1){
//a, b 為區間修改的left and right, cmd 為要增加的數值
    if(a <= node[x].l && b >= node[x].r){
        //如果節點的left and right，跟a, b 區間是相等，或更小就，只要在這邊修改cmd，
        //就可以讓node[x].v 的值直接變為區間修改後的數值，
        //之後如果要讓這查詢向子樹進行區間修改，就用push_down，
        //我們這邊的懶人標記就會告訴左右子樹要修改的值為多少

        node[x].v += cmd; //區間修改後的v
        node[x].z = cmd; //區間修改是要增加多少數值
        return;
    }
    push_down(x);//先將之前的區間查詢修改值，往下給子樹以避免上次的查詢值被忽略
    //假如當前的node[x].z 原本是3，如果沒有push_down(x)，那下面的子樹都沒有被+3，
    //導致答案不正確。

    int mid = (node[x].l+node[x].r) / 2;  //切半，向下修改
    if(a <= mid) update(a, b, cmd, Lson(x)); //如果要修改的點在左邊，就往左下角追蹤
    if(b > mid) update(a, b, cmd, Rson(x)); //如果要修改的點在右邊，就往右下角追蹤
    node[x].v = node[Lson(x)].v + node[Rson(x)].v;
    //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
```

8.3 Suffix Automaton

```
// 只要關於這兩個字串問題都可以使用O(n) 時間複雜度解決：
// 在另一個字串中查詢另一個字串的所有出現位置
// 計算此字串中裡面有多少不同的子字串

// 需要用到struct，此struct 需要len , link , next，這些的意義為：

// len 目前的最長長度
// link 為當前子字串中第一個最長後綴結束位置
// next 連結其他的點的邊，方向是->

// 重大的三個特性
// 跟著藍色線走到終點時會是必定是\aabbabd" 的後綴
// 跟著藍色線走到任意點必定會是此字串的子字串
// 發明這個的演算法大師太強了，跟神一般的存在

#define SAMN N*10
// N 為字串最長長度
int sz , last ; // 到SAM 初始化說明

struct state{
    int len , link ; // len = 最長長度, link = 當前子字串中第一個最長後綴結束位置

    map<char,int> next ;
}st[SAMN];

void sam_init(){
    sz = 0 ;
    st[0].len = 0 ;
    st[0].link = -1 ;
    st[0].next.clear();
    sz++ ;
    last = 0 ;
}

void sam_extend(char c ){ //char c 要擴增的字元
    int cur = sz++ ; //sz++ 增加sam array 長度, cur 為當前的sam 節點
    st[cur].next.clear() ; //先把當前的sam 連接點狀態移除
    st[cur].len = st[last].len+1 ; //為前一個sam 節點len +1 表示其長度
    int p = last ; // p = 查詢當前字串的「所有子字串」與新增加c 後的字串是否有共同後綴，
    //將跑到他們有共同後綴的「前一個位置」
    //注意：這裡的共同後綴只要有一個字元是就可以是共同後綴
    //舉例："abca" and "abcab" 中的'b' 就是共同後綴

    while(p != -1 && !st[p].next.count(c)){ // p = -1 表示已經到起點，
        // !st[p].next.count(c) 則是詢問增加此字元後是否會有共同後綴的情形，
        // 如果有則需要額外處理
        st[p].next[c] = cur ; // 將前面的點與現在的sam 節點做連結
        p = st[p].link ; // 由於現在的字元並沒有和前面的子字串有共同後綴，
        // 於是他們的link 就向上追蹤
        // 如果有則st[p].next.count(c) == TRUE 不符合迴圈要求
    }
    if(p == -1){
        // p = -1 表示沒有共同後綴且此字元在當前字串中從沒出現過，
        //才回到了起始點，所以將link 設置為0
        st[cur].link = 0 ;
    }
    else{
        int q = st[p].next[c] ; // q 為他們共同後綴的位置
        if(st[p].len + 1 == st[q].len){
            //如果st[p].len + 1 == st[q].len 表示「不同位置但相同字元」的共同後綴長度大於一
            //只需要直接將當前的sam[cur].link 設定成q 也就是共同後綴的位置
            st[cur].link = q ;
        }
        else{ // 如果不同位置但相同字元的共同後綴如果等於一，則需要連創建新的sam 節點，
            // 建立以c + 字串前一個字元的後綴(前一個並不包括我們現在新增的c)，
            // 並同時放棄另一個不同位置但也是c 字元的後綴，但要持續存在以保護先前做好的sam
            int clone = sz++ ; // 創建新節點
            st[clone].len = st[p].len + 1 ; // 表示從共同後綴的前一個位置+1，
            //用來建立以c + 字串前一個字元的後綴
            st[clone].next = st[q].next ; //複製q 的next，因為前面已經設定好連接的點，
            //但是因為共同後綴不同，後面還需要一個while 迴圈進行調整
            st[clone].link = st[q].link ; //將他們link 先設置相同，
            //之後用while 迴圈再移動到正確的link
            while(p != -1 && st[p].next[c] == q){
                //p != -1 是不可以讓她更改起始點的位置
                //st[p].next[c] == q 接下來的點是從clone 繼續擴展而不是原先的q，
                //所以要將原先連接到q 的點全部改連接至clone
                st[p].next[c] = clone ; //更改連接點至clone
                p = st[p].link ; // 繼續往上層追蹤
            }
            st[q].link = st[cur].link = clone ;
            // 最後則是也要把q and cur 的link 改到clone，
            // 原因則是因為接下來的點是從clone 繼續擴展而不是原先的q
        }
    }
    last = cur ; //準備下一次的擴展
}

// QUESTION: 最小循環移位(Lexicographically minimum string rotation) 是甚麼？
// 給你一組字串，找出字典序最小的循環字串，沒錯，就是這題的題目，非常純粹的模板題。

// 要怎麼解開呢？
// 其實容易想到，只需要將原本的字串複製一次給原本的字串，即string += string，透過從起始點一路跟著當下可以走的最小字典序
    節點走，走到原先字串的長度，在k-string.length()+1，就是最小循環移位了。

// QUESTION A: 為甚麼只要原本的字串複製一次給原本的字串呢？
// 由於第一次的字串長度結束位置+ 字串長度(即第二次循環) < 連續三次循環長度，就算從最後一個字元開始循環也不會大於三次循
    環，即可證明我們不需要第三次循環，只要循環一次就好。

//st 是sam、now 是還要再找幾次，一開始為原本字串長度
while(now--){
    for(auto it : st[u].next){ //跟著字典續追蹤
        u = it.second ;
        break ; //找到了就往下個節點移動，類似於DFS
    }
}
cout << st[u].len - len + 1 << '\n' ;
//找到當下的節點後，找出它的長度並且扣掉原始長度並加一即是答案
```

8.4 suffix tree

```
// 以下是Suffix Tree 能解決的問題：

// 尋找A 字串是否在字串B 中
// 找出B 在A 字串重複的次數
// 最長共同子字串

// 時間複雜度O(n)

// remaining 隱藏在Suffix Tree 中的後綴節點
// root = Suffix Tree 的最主要根節點
// active_node 活動節點，主要是用來生長葉節點(leaf)
// active_e 隱藏節點的第一個字元
// active_len 隱藏在Suffix Tree 中節點的長度
```

```
// node 一個struct 用來存入Suffix Tree 節點
// start 此節點開始的位置(index)
// end 此節點結束的位置(end)
// 舉例：node.start = 3 and node.end = 5，則string 的長度是string.substr(3,2)，用數學表示則是(start,end]
// next 用來指出下一個節點的位置，個人習慣用map
// slink 指出此節點的最長後綴節點，EX: XYZ 則指出YZ。
// edge_length() 公式為min(end,pos+1)start
// 用來找出此節點的字串長度

struct node{
    int start , end ,slink ;
    map<char,int> next ;

    int edge_length(){
        return min(end , pos+1) - start ;
    }

    void init(int st , int ed = oo){
        start = st ;
        end = ed ;
        slink = 0 ;
        next.clear() ;
    }
}tree[2*N];

void st_init(){
    //tree root is 1 not zero
    needSL = remainder_ = 0 ;
    active_node = active_e = active_len = 0 ;
    pos = -1 ;

    cnt = root = 1 ;
    active_node = 1 ;
    tree[cnt++].init(-1,-1);
    return ;
}

char active_edge(){ //隱藏字元的第一個
    return text[active_e] ;
}

void add_SL(int node){ // slink 指回上一個隱藏節點的位置，如果上一個後綴節點的葉節點需要被更改時，
// 這裡的下方葉節點也能被迅速被更改，達到O(1) 效果
    if(needSL > 0 ) tree[needSL].slink = node ;
    needSL = node ;
}

bool walkdown(int node){ //即原理說明 "xyzxyaxyz$" 的step 1，xyz 但xy 是一個節點，
// 需要在往下一個子節點前進
    if(active_len >= tree[node].edge_length()){
        active_e += tree[node].edge_length() ; //找到此長度後的第一個隱藏字元
        active_len -= tree[node].edge_length() ; //減少長度
        active_node = node ; //往後方前進
        return true ;
    }
    return false ;
}

void st_extend(char c){ //擴增suffix tree
    pos++; // 往下個字串前進
    needSL = 0 ; // 紀錄上一個切割點的位置，用來slink 的前一個點
    remainder_++ ; // 先+1，如果這個點有被增加之後做-1 的動作
    while(remainder_ > 0){
        if(active_len == 0 ) active_e = pos ;
        // 如果active len 等於0，就表示沒有隱藏長度，所以我們要判斷的就是當前字元
        // 是否存在此active_node 節點中
        if(tree[active_node].next[active_edge()] == 0){
            // active_node 沒有此字元的節點，新增節點
            int leaf = cnt ;
            tree[cnt++].init(pos) ;
            tree[active_node].next[active_edge()] = leaf ;
            add_SL(active_node)   ;// 紀錄slink 的位置，以防下次用到
        }
        else{ // active_node 有此字元的節點
            int nxt = tree[active_node].next[active_edge()] ;
            if(walkdown(nxt)) continue ; // 如果還需要在往下一個節點走，就減少隱藏長度，
            //然後回去重新查詢
            if(text[tree[nxt].start + active_len] == c){
                // 如果此節點有包含到此字元，代表隱藏長度可以+1，因為後綴還是在節點長裡面
                active_len++ ; // 隱藏長度可以+1
                add_SL(active_node) ; // 紀錄slink 的位置，以防下次用到
                break ; //由於隱藏節點是+1，所以我們沒必要減
            }
            // 需要做切割點
            int split = cnt ;
            tree[cnt++].init(tree[nxt].start , tree[nxt].start + active_len) ;
            //製作切割點中...，結束位置就是當前節點的start + 隱藏長度
            tree[active_node].next[active_edge()] = split ;
            // 需要將active_node 指向我們的切割點，而不是原來的點
            int leaf = cnt ; // 需要葉節點
            tree[cnt++].init(pos) ;
            // 製作葉節點
            tree[split].next[c] = leaf ; // 把葉節點指向我們的切割點
            tree[nxt].start += active_len ; //原本的節點start 往後到切割點的end
            tree[split].next[text[tree[nxt].start]] = nxt ; //將原本節點指向我們的切割點
            add_SL(split) ; // 紀錄slink 的位置，以防下次用到
        }
        remainder_-- ; // 由於有增加節點，所以-1
        if(active_node == root && active_len > 0 ){
            //active_len > 0 表示我們現在做的是把隱藏節點新增，所以要減掉
            //active_node == root 確保有回到根節點才做隱藏節點減掉，否則
            //text[node.start + active_len ] 就會亂掉
            active_len-- ;
            //由於我們減少了一個隱藏長度，所以-1
            active_e = pos - remainder_ + 1 ;
            //找到減少後隱藏長度的第一個隱藏字元，此時如果active_len == 0，
            // 則下次迴圈則在active_e 會被重新定義成pos
        }
        else{
            // 跟著slink 走去改動其他的後綴在tree[active_node].slink > 0 時，
            // 否則則回到root，繼續建立後綴樹
            active_node = tree[active_node].slink > 0 ? tree[active_node].slink : root ;
        }
    }
    return ;
}
```