

# Algorithm Library

Liu Yang

December 22, 2018

## Contents

<b>1</b>	<b>String</b>	<b>4</b>
1.1	AhoCorasickAutomaton . . . . .	4
1.2	KMP . . . . .	5
1.3	Manacher . . . . .	7
1.4	PalindromicTree . . . . .	8
<b>2</b>	<b>Math</b>	<b>10</b>
2.1	Catalan . . . . .	10
2.2	Derangement . . . . .	10
2.3	Euler . . . . .	11
2.3.1	Euler . . . . .	11
2.3.2	PrimeEuler . . . . .	11
2.3.3	Sieve . . . . .	12
2.4	FFT . . . . .	13
2.5	Fibonacci . . . . .	14
2.6	GeneratingFunction . . . . .	16
2.7	InverseElement . . . . .	16
2.7.1	ExtendGcd . . . . .	16
2.7.2	Factorial . . . . .	17
2.7.3	FermatLittleTheorem . . . . .	18
2.7.4	Recursive . . . . .	18
2.8	Prime . . . . .	19
2.8.1	PrimeFactor . . . . .	19
2.8.2	SieveOfEratosthenes . . . . .	19
2.9	QuickPow . . . . .	20
2.10	Stirling . . . . .	21
<b>3</b>	<b>DataStructure</b>	<b>22</b>
3.1	BinaryIndexedTree . . . . .	22
3.2	DfsOrder . . . . .	22
3.3	SegmentTree . . . . .	23
3.3.1	SegmentTree . . . . .	23
3.3.2	SegmentTreestruct . . . . .	26
3.4	Splay . . . . .	29
3.4.1	SplayTree . . . . .	29
3.4.2	SplayTreeArray . . . . .	33
3.5	TrieTree . . . . .	37
<b>4</b>	<b>GraphTheory</b>	<b>39</b>
4.1	LCA . . . . .	39
4.1.1	DFS+ST . . . . .	39
4.1.2	Tarjan . . . . .	41

4.2	MinimumSpanningTree	43
4.2.1	Kruskal	43
4.2.2	Prim	45
4.3	NetworkFlow	46
4.3.1	Dinic	46
4.3.2	FordFulkerson	49
4.3.3	MinCostMaxFlow	50
4.4	ShortestPath	52
4.4.1	BellmanFord	52
4.4.2	Dijkstra	53
4.4.3	Floyd	55
4.4.4	SPFA	55
<b>5</b>	<b>DynamicProgramming</b>	<b>58</b>
5.1	Contour	58
5.2	Digit	58
5.3	LCS	59
5.4	LIS	60
5.5	Pack	61
<b>6</b>	<b>ComputationalGeometry</b>	<b>62</b>
6.1	Plane	62
6.2	Stereoscopic	68
<b>7</b>	<b>Others</b>	<b>70</b>
7.1	Factorial	70
7.2	FastIO	71
7.3	LeapYear	75
7.4	NimGame	75
7.5	vim	76

## 1 String

### 1.1 AhoCorasickAutomaton

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

const int maxn = "Edit";

struct AhoCorasickAutomaton {
    // 子节点记录数组
    int Son[maxn][26];
    int Val[maxn];
    // 失配指针 Fail 数组
    int Fail[maxn];
    // 节点数量
    int Tot;

    // Trie Tree 初始化
    void TrieInit() {
        Tot = 0;
        memset(Son, 0, sizeof(Son));
        memset(Val, 0, sizeof(Val));
        memset(Fail, 0, sizeof(Fail));
    }

    // 计算字母下标
    int Pos(char X) {
        return X - 'a';
    }

    // 向 Trie Tree 中插入 Str 模式字符串
    void Insert(string Str) {
        int Cur = 0, Len = int(Str.length());
        for (int i = 0; i < Len; ++i) {
            int Index = Pos(Str[i]);
            if (!Son[Cur][Index]) {
                Son[Cur][Index] = ++Tot;
            }
            Cur = Son[Cur][Index];
        }
        Val[Cur]++;
    }

    // Bfs 求得 Trie Tree 上失配指针
```

```
void GetFail() {
    std::queue<int> Que;
    for (int i = 0; i < 26; ++i) {
        if (Son[0][i]) {
            Fail[Son[0][i]] = 0;
            Que.push(Son[0][i]);
        }
    }
    while (!Que.empty()) {
        int Cur = Que.front(); Que.pop();
        for (int i = 0; i < 26; ++i) {
            if (Son[Cur][i]) {
                Fail[Son[Cur][i]] = Son[Fail[Cur]][i];
                Que.push(Son[Cur][i]);
            }
            else {
                Son[Cur][i] = Son[Fail[Cur]][i];
            }
        }
    }
}

// 询问 Str 中出现的模式串数量
int Query(string Str) {
    int Len = int(Str.length());
    int Cur = 0, Ans = 0;
    for (int i = 0; i < Len; ++i) {
        Cur = Son[Cur][Pos(Str[i])];
        for (int j = Cur; j && ~Val[j]; j = Fail[j]) {
            Ans += Val[j];
            Val[j] = -1;
        }
    }
    return Ans;
}

};
```

## 1.2 KMP

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

```
// 对模式串 Pattern 计算 Next 数组
```

```
void KMPPre(string Pattern, vector<int> &Next) {
    int i = 0, j = -1;
```

```
Next[0] = -1;
int Len = int(Pattern.length());
while (i != Len) {
    if (j == -1 || Pattern[i] == Pattern[j]) {
        Next[++i] = ++j;
    }
    else {
        j = Next[j];
    }
}
}

// 优化对模式串 Pattern 计算 Next 数组
void PreKMP(string Pattern, vector<int> &Next) {
    int i, j;
    i = 0;
    j = Next[0] = -1;
    int Len = int(Pattern.length());
    while (i < Len) {
        while (j != -1 && Pattern[i] != Pattern[j]) {
            j = Next[j];
        }
        if (Pattern[++i] == Pattern[++j]) {
            Next[i] = Next[j];
        }
        else {
            Next[i] = j;
        }
    }
}

// 利用预处理 Next 数组计数模式串 Pattern 在主串 Main 中出现次数
int KMPCount(string Pattern, string Main) {
    int PatternLen = int(Pattern.length()), MainLen =
        ↪ int(Main.length());
    vector<int> Next(PatternLen + 1, 0);
    //PreKMP(Pattern, Next);
    KMPPre(Pattern, Next);
    int i = 0, j = 0;
    int Ans = 0;
    while (i < MainLen) {
        while (j != -1 && Main[i] != Pattern[j]) {
            j = Next[j];
        }
```

```
    }
    i++; j++;
    if (j >= PatternLen) {
        Ans++;
        j = Next[j];
    }
}
return Ans;
}
```

### 1.3 Manacher

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

const int maxn = "Edit";

char ConvertStr[maxn << 1];
int Len[maxn << 1];

// Manacher 算法求 Str 字符串最长回文子串长度
int Manacher(char Str[]) {
    int L = 0, StrLen = int(strlen(Str));
    ConvertStr[L++] = '$'; ConvertStr[L++] = '#';
    for (int i = 0; i < StrLen; ++i) {
        ConvertStr[L++] = Str[i];
        ConvertStr[L++] = '#';
    }
    int MX = 0, ID = 0, Ans = 0;
    for (int i = 0; i < L; ++i) {
        Len[i] = MX > i ? min(Len[2 * ID - i], MX - i) : 1;
        while (ConvertStr[i + Len[i]] == ConvertStr[i -
        ↪ Len[i]]) {
            Len[i]++;
        }
        if (i + Len[i] > MX) {
            MX = i + Len[i];
            ID = i;
        }
        Ans = max(Ans, Len[i] - 1);
    }
    return Ans;
}
```

## 1.4 PalindromicTree

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

const int maxn = "Edit";

struct PalindromicTree {
    // 子节点记录数组
    long long Son[maxn][26];
    // 失配指针 Fail 数组
    long long Fail[maxn];
    // Len[i]: 节点 i 表示的回文串长度 (一个节点表示一个回文串)
    long long Len[maxn];
    // Cnt[i]: 节点 i 表示的本质不同的串的个数 (最后需要运行
    //    $\hookrightarrow$  Count() 函数才可求出正确结果)
    long long Cnt[maxn];
    // Num[i]: 以节点 i 表示的最长回文串的最右端为回文串结尾的回
    //    $\hookrightarrow$  文串个数
    long long Num[maxn];
    // 字符
    long long Str[maxn];
    // 新添加字符后最长回文串表示的节点
    long long Last;
    // 字符数量
    long long StrLen;
    // 节点数量
    long long Tot;

    // 新建节点
    long long NewNode(long long X) {
        for (long long i = 0; i < 26; ++i) {
            Son[Tot][i] = 0;
        }
        Cnt[Tot] = 0;
        Num[Tot] = 0;
        Len[Tot] = X;
        return Tot++;
    }

    // 初始化
    void Init() {
        Tot = 0;
        NewNode(0); NewNode(-1);
        Last = 0;
    }
};
```



```
    StrLen = 0;
    // 开头存字符集中没有的字符，减少特判
    Str[0] = -1;
    Fail[0] = 1;
}

long long GetFail(long long X) {
    while (Str[StrLen - Len[X] - 1] != Str[StrLen]) {
        X = Fail[X];
    }
    return X;
}

void Add(long long Char) {
    Char -= 'a';
    Str[++StrLen] = Char;
    long long Cur = GetFail(Last);
    if (!Son[Cur][Char]) {
        long long New = NewNode(Len[Cur] + 2);
        Fail[New] = Son[GetFail(Fail[Cur])][Char];
        Son[Cur][Char] = New;
        Num[New] = Num[Fail[New]] + 1;
    }
    Last = Son[Cur][Char];
    Cnt[Last]++;
}

void Count() {
    // 若 Fail[V]=U, 则 U 一定是 V 回文子串，所以双亲累加孩
    //   子的 Cnt
    for (long long i = Tot - 1; i >= 0; --i) {
        Cnt[Fail[i]] += Cnt[i];
    }
}

};
```

## 2 Math

### 2.1 Catalan

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

const int maxn = "Edit";

long long Catalan[maxn];

// 递推求卡特兰数
void CalalanInit() {
    memset(Catalan, 0, sizeof(Catalan));
    Catalan[0] = Catalan[1] = 1;
    for (int i = 2; i < maxn; ++i) {
        Catalan[i] = Catalan[i - 1] * (4 * i - 2) / (i + 1);
    }
}
```

### 2.2 Derangement

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

const int maxn = "Edit";
const int mod = 1e9 + 7;

// Staggered: 错排数
long long Staggered[maxn];

// 求错排数
void StaggeredInit() {
    Staggered[1] = 0;
    Staggered[2] = 1;
    // 递推求错排数
    for (int i = 3; i < maxn; ++i) {
        Staggered[i] = (i - 1) * (Staggered[i - 1] +
            ↪ Staggered[i - 2]) % mod;
    }
}
```

## 2.3 Euler

### 2.3.1 Euler

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

// 单独求解欧拉函数
int Phi(int X) {
    int Ans = X;
    for (int i = 2; i * i <= X; ++i) {
        if (!(X % i)) {
            Ans = Ans / i * (i - 1);
            while (!(X % i)) {
                X /= i;
            }
        }
    }
    if (X > 1) {
        Ans = Ans / X * (X - 1);
    }
    return Ans;
}
```

### 2.3.2 PrimeEuler

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

const int maxn = "Edit";

// 素数标记
bool IsPrime[maxn];
// 欧拉函数
int Phi[maxn];
// 素数
int Prime[maxn];
// 素数个数
int Tot;

// 同时求得欧拉函数和素数表
void PhiPrime() {
    memset(IsPrime, false, sizeof(IsPrime));
    Phi[1] = 1;
    Tot = 0;
    for (int i = 2; i < maxn; ++i) {
        if (!IsPrime[i]) {
```

```
        Prime[Tot++] = i;
        Phi[i] = i - 1;
    }
    for (int j = 0; j < Tot; ++j) {
        if (i * Prime[j] > maxn) {
            break;
        }
        IsPrime[i * Prime[j]] = true;
        if (!(i % Prime[j])) {
            Phi[i * Prime[j]] = Phi[i] * Prime[j];
            break;
        }
        else {
            Phi[i * Prime[j]] = Phi[i] * (Prime[j] - 1);
        }
    }
}
}
```

### 2.3.3 Sieve

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

const int maxn = "Edit";

// 欧拉函数
int Phi[maxn];

// 筛法求欧拉函数
void Euler() {
    for (int i = 1; i < maxn; ++i) {
        Phi[i] = i;
    }
    for (int i = 2; i < maxn; i += 2) {
        Phi[i] /= 2;
    }
    for (int i = 3; i < maxn; i += 2) {
        if (Phi[i] == i) {
            for (int j = i; j < maxn; j += i) {
                Phi[j] = Phi[j] / i * (i - 1);
            }
        }
    }
}
```

## 2.4 FFT

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

const int maxn = "Edit";
const double pi = acos(-1.0);

// 复数
struct Complex {
    double X, Y;

    Complex operator + (const Complex &B) const {
        return Complex {X + B.X, Y + B.Y};
    }

    Complex operator - (const Complex &B) const {
        return Complex {X - B.X, Y - B.Y};
    }

    Complex operator * (const Complex &B) const {
        return Complex {X * B.X - Y * B.Y, X * B.Y + Y * B.X};
    }

    Complex operator / (const Complex &B) const {
        double Temp = B.X * B.X + B.Y * B.Y;
        return Complex {(X * B.X + Y * B.Y) / Temp, (Y * B.X -
            ↪ X * B.Y) / Temp};
    }
};

// 多项式系数数量
int N, M;
int L;
int Limit;
int R[maxn << 2];

// 快速傅里叶变换 (FFT)
void FFT(Complex F[], int Op) {
    for (int i = 0; i < Limit; ++i) {
        if (i < R[i]) {
            std::swap(F[i], F[R[i]]);
        }
    }
    for (int j = 1; j < Limit; j <= 1) {
```

```
Complex Temp = Complex {cos(pi / j), Op * sin(pi /
↪ j)};
for (int k = 0; k < Limit; k += (j << 1)) {
    Complex Buffer = Complex {1.0, 0.0};
    for (int l = 0; l < j; ++l) {
        Complex Tx = F[k + l], Ty = Buffer * F[k + j +
↪ l];
        F[k + l] = Tx + Ty;
        F[k + j + l] = Tx - Ty;
        Buffer = Buffer * Temp;
    }
}
```

// 多项式系数

```
Complex A[maxn], B[maxn];
```

// 多项式卷积计算

```
void Cal() {
    Limit = 1; L = 0;
    while (Limit <= N + M) {
        Limit <<= 1;
        L++;
    }
    for (int i = 0; i < Limit; ++i) {
        R[i] = (R[i >> 1] >> 1) | ((i & 1) << (L - 1));
    }
    FFT(A, 1);
    FFT(B, 1);
    for (int i = 0; i <= Limit; ++i) {
        A[i] = A[i] * B[i];
    }
    FFT(A, -1);
}
```

## 2.5 Fibonacci

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

```
const int mod = 1e9 + 7;
```

// 矩阵结构体

```
struct Matrix {
```

```
// 矩阵
long long Mat[2][2];
Matrix() {}
// 重载矩阵乘法
Matrix operator * (Matrix const &A) const {
    Matrix Res;
    memset(Res.Mat, 0, sizeof(Res.Mat));
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            for (int k = 0; k < 2; ++k) {
                Res.Mat[i][j] = (Res.Mat[i][j] + Mat[i][k]
                    ↪ * A.Mat[k][j] % mod) % mod;
            }
        }
    }
    return Res;
}

};

// 重载矩阵快速幂
Matrix operator ^ (Matrix Base, long long K) {
    Matrix Res;
    memset(Res.Mat, 0, sizeof(Res.Mat));
    Res.Mat[0][0] = Res.Mat[1][1] = 1;
    while (K) {
        if (K & 1) {
            Res = Res * Base;
        }
        Base = Base * Base;
        K >>= 1;
    }
    return Res;
}

// 斐波那契数列中第 x 项
long long Fib(long long X) {
    Matrix Base;
    Base.Mat[0][0] = Base.Mat[1][0] = Base.Mat[0][1] = 1;
    Base.Mat[1][1] = 0;
    return (Base ^ X).Mat[0][1];
}
```

## 2.6 GeneratingFunction

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

const int maxn = "Edit";

void GeneratingFunction() {
    int n;
    int c1[maxn], c2[maxn];
    scanf("%d", &n);
    for (int i = 0; i < maxn; ++i) {
        c1[i] = 1;
        c2[i] = 0;
    }
    // c1[i] 为  $x^i$  的系数
    // c2 为中间变量
    for (int i = 2; i <= n; ++i) {
        for (int j = 0; j <= n; ++j) {
            for (int k = 0; k + j <= n; k += i) {
                c2[j + k] += c1[i];
            }
        }
        for (int j = 0; j <= n; ++j) {
            c1[j] = c2[j];
            c2[j] = 0;
        }
    }
}
```

## 2.7 InverseElement

### 2.7.1 ExtendGcd

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

// 扩展欧几里得,  $A*X+B*Y=D$ 
long long ExtendGcd(long long A, long long B, long long &X,
    ↪ long long &Y) {
    // 无最大公约数
    if (A == 0 && B == 0) {
        return -1;
    }
    if (B == 0) {
        X = 1;
        Y = 0;
    }
```



```

        return A;
    }
    long long D = ExtendGcd(B, A % B, Y, X);
    Y -= A / B * X;
    return D;
}

```

```

// 逆元,  $AX = 1(mod M)$ 
long long Inv(long long A, long long N) {
    long long X, Y;
    long long D = ExtendGcd(A, N, X, Y);
    if (D == 1) {
        return (X % N + N) % N;
    }
    else {
        return -1;
    }
}

```

### 2.7.2 Factorial

```

#include <bits/stdc++.h>

const int mod = 1e9 + 7;
const int maxn = "Edit";

// 快速乘
long long QuickMul(long long A, long long B) {
    long long Ans = 0;
    while (B) {
        if (B & 1) {
            Ans = (Ans + A) % mod;
        }
        A = (A + A) % mod;
        B >>= 1;
    }
    return Ans;
}

```

```

// 快速幂
long long QuickPow(long long A, long long B) {
    long long Ans = 1;
    while (B) {
        if (B & 1) {

```

```

        Ans = QuickMul(Ans, A) % mod;
    }
    A = QuickMul(A, A) % mod;
    B >>= 1;
}
return Ans;
}

// Factorial: 阶乘, FactorialInv: 阶乘逆元
long long Factorial[maxn], FactorialInv[maxn];

// 求阶乘逆元
void FactorialInvInit() {
    // 求阶乘
    Factorial[0] = 0;
    Factorial[1] = 1;
    for (int i = 2; i < maxn; ++i) {
        Factorial[i] = (Factorial[i - 1] * i) % mod;
    }
    // 飞马小定理求最大值阶乘逆元
    FactorialInv[maxn - 1] = QuickPow(Factorial[maxn - 1], mod
    ↪ - 2);
    // 递推求阶乘逆元
    for (int i = maxn - 2; i >= 0; --i) {
        FactorialInv[i] = (FactorialInv[i + 1] * (i + 1)) %
        ↪ mod;
    }
}

```

### 2.7.3 FermatLittleTheorem

```

#include <bits/stdc++.h>

const int mod = 1e9 + 7;

// 快速幂、费马小定理求逆元
long long Inv(long long X) {
    return QuickPow(X, mod - 2);
}

```

### 2.7.4 Recursive

```

#include <bits/stdc++.h>

```

```

const int mod = 1e9 + 7;
const int maxn = "Edit";

long long Inv[maxn];

// 递推求逆元
void InvInit() {
    Inv[1] = 1;
    for (int i = 2; i < maxn; ++i) {
        Inv[i] = (mod - mod / i) * Inv[mod % i] % mod;
    }
}

```

## 2.8 Prime

### 2.8.1 PrimeFactor

```

#include <bits/stdc++.h>

const int maxn = "Edit"

bool IsPrime[maxn];
vector<int> PrimeFactor[maxn];

void Init() {
    memset(IsPrime, true, sizeof(IsPrime));
    for (long long i = 2; i < maxn; ++i) {
        if (IsPrime[i]) {
            PrimeFactor[i].push_back(i);
            for (long long j = i + i; j < maxn; ++j) {
                IsPrime[j] = false;
                PrimeFactor[j].push_back(i);
            }
        }
    }
    IsPrime[1] = false;
}

```

### 2.8.2 SieveOfEratosthenes

```

#include <bits/stdc++.h>

const int maxn = "Edit";

bool IsPrime[maxn];

```

```
void Init() {
    memset(IsPrime, true, sizeof(IsPrime));
    IsPrime[0] = IsPrime[1] = false;
    for (long long i = 2; i < maxn; ++i) {
        if (IsPrime[i]) {
            for (long long j = i * i; j < maxn; j += i) {
                IsPrime[j] = false;
            }
        }
    }
}
```

## 2.9 QuickPow

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

const int mod = 1e9 + 7;

// 快速乘求  $A*B\%mod$ 
long long QuickMul(long long A, long long B) {
    long long Ans = 0;
    while (B) {
        if (B & 1) {
            Ans = (Ans + A) % mod;
        }
        A = (A + A) % mod;
        B >>= 1;
    }
    return Ans;
}

// 快速幂求  $A^B\%mod$ 
long long QuickPow(long long A, long long B) {
    long long Ans = 1;
    while (B) {
        if (B & 1) {
            Ans = QuickMul(Ans, A) % mod;
        }
        A = QuickMul(A, A) % mod;
        B >>= 1;
    }
    return Ans;
}
```

## 2.10 Stirling

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

const double pi = acos(-1.0);
const double e = 2.718281828459;

int Stirling(int x) {
    if (x <= 1) {
        return 1;
    }
    return int(ceil(log10(2 * pi * x) / 2 + x * log10(x /
        ↪ e))));
}
```

## 3 DataStructure

### 3.1 BinaryIndexedTree

```
#include <bits/stdc++.h>
#define lowbit(x) (x&(-x))

const int maxn = "Edit";

// 树状数组
int C[maxn];

// 更新树状数组信息
void Update(int X, int Val) {
    while (X < maxn) {
        C[X] += Val;
        X += lowbit(X);
    }
}

// 求和
int GetSum(int X) {
    int Res = 0;
    while (X > 0) {
        Res += C[X];
        X -= lowbit(X);
    }
    return Res;
}
```

### 3.2 DfsOrder

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

const int maxn = "Edit";

// 链式前向星建图
struct Link {
    int V, Next;
};

Link edges[maxn << 1];
int Head[maxn];
int Tot = 0;
```

```

void Init() {
    Tot = 0;
    memset(Head, -1, sizeof(Head));
}

void AddEdge(int U, int V) {
    edges[++Tot] = Link {V, Head[U]};
    Head[U] = Tot;
    edges[++Tot] = Link {U, Head[V]};
    Head[V] = Tot;
}

int Cnt;
int InIndex[maxn], OutIndex[maxn];

// Dfs 序
void DfsSequence(int Node, int Pre) {
    Cnt++;
    InIndex[Node] = Cnt;
    for (int i = Head[U]; i != -1; i = edges[i].Next) {
        if (edges[i].V != Pre) {
            DfsSequence(edges[i].V, Node);
        }
    }
    OutIndex[U] = Cnt;
}

```

### 3.3 SegmentTree

#### 3.3.1 SegmentTree

```

#include <bits/stdc++.h>

const int maxn = "Edit";

// Sum: 线段树信息 (此模板为求和), Lazy: 惰性标记
int Sum[maxn << 2], Lazy[maxn << 2];

// 更新节点信息, 这里是求和
void PushUp(int Root) {
    Sum[Root] = Sum[Root << 1] + Sum[Root << 1 | 1];
}

// 下推标记函数, LeftNum, RightNum: 分别为左右子树的数字数量

```

```

void PushDown(int Root, int LeftNum, int RightNum) {
    if (Lazy[Root]) {
        // 下推标记
        Lazy[Root << 1] += Lazy[Root];
        Lazy[Root << 1 | 1] += Lazy[Root];
        // 根据惰性标修改子节点的值
        Sum[Root << 1] += Lazy[Root] * LeftNum;
        Sum[Root << 1 | 1] += Lazy[Root] * RightNum;
        // 清除本节点惰性标记
        Lazy[Root] = 0;
    }
}

// 建树, Left、Right: 当前节点区间, Root: 当前节点编号
void Build(int Left, int Right, int Root) {
    Lazy[Root] = 0;
    // 到达叶子节点
    if (Left == Right) {
        scanf("%d", &Sum[Root]);
        return;
    }
    int Mid = (Left + Right) >> 1;
    // 左子树
    Build(Left, Mid, Root << 1);
    // 右子树
    Build(Mid + 1, Right, Root << 1 | 1);
    // 更新信息
    PushUp(Root);
}

// 单点修改, Pos: 修改点位置, Value: 修改值, Left、Right: 当前区
↪ 间, Root: 当前节点编号
void PointUpdate(int Pos, int Value, int Left, int Right, int
↪ Root) {
    // 修改叶子节点
    if (Left == Right) {
        Sum[Root] += Value;
        return;
    }
    int Mid = (Left + Right) >> 1;
    // 根据条件判断调用左子树还是右子树
    if (Pos <= Mid) {
        PointUpdate(Pos, Value, Left, Mid, Root << 1);
    }
}

```



```

    else {
        PointUpdate(Pos, Value, Mid + 1, Right, Root << 1 |
            ↪ 1);
    }
    // 子节点更新后更新此节点
    PushUp(Root);
}

// 区间修改, OperateLeft、OperateRight: 操作区间, Left、Right:
↪ 当前区间, Root: 当前节点编号
void IntervalUpdate(int OperateLeft, int OperateRight, int
    ↪ Value, int Left, int Right, int Root) {
    // 若本区间完全在操作区间内
    if (OperateLeft <= Left && OperateRight >= Right) {
        Sum[Root] += Value * (Right - Left + 1);
        // 增加惰性标记, 表示本区间 Sum 正确, 但子区间仍需要根据
        ↪ 惰性标记调整更新
        Lazy[Root] += Value;
        return;
    }
    int Mid = (Left + Right) >> 1;
    // 下推标记
    PushDown(Root, Mid - Left + 1, Right - Mid);
    // 根据条件判断调用左子树还是右子树
    if (OperateLeft <= Mid) {
        IntervalUpdate(OperateLeft, OperateRight, Value, Left,
            ↪ Mid, Root << 1);
    }
    if (OperateRight > Mid) {
        IntervalUpdate(OperateLeft, OperateRight, Value, Mid +
            ↪ 1, Right, Root << 1 | 1);
    }
    // 更新当前节点信息
    PushUp(Root);
}

// 区间查询, OperateLeft、OperateRight: 操作区间, Left、Right:
↪ 当前区间, Root: 当前节点编号
int Query(int OperateLeft, int OperateRight, int Left, int
    ↪ Right, int Root) {
    // 区间内直接返回
    if (OperateLeft <= Left && OperateRight >= Right) {
        return Sum[Root];
    }

```

```
}
int Mid = (Left + Right) >> 1;
// 下推标记
PushDown(Root, Mid - Left + 1, Right - Mid);
// 叠加结果
int Ans = 0;
if (OperateLeft <= Mid) {
    Ans += Query(OperateLeft, OperateRight, Left, Mid,
        ↪ Root << 1);
}
if (OperateRight > Mid) {
    Ans += Query(OperateLeft, OperateRight, Mid + 1,
        ↪ Right, Root << 1 | 1);
}
// 返回结果
return Ans;
}
```

### 3.3.2 SegmentTreestruct

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

const int maxn = "Edit";

// 线段树节点
struct Node {
    int Left, Right;
    int Lazy, Tag;
    int Sum;
};

Node SegmentTree[maxn << 2];

// 更新节点信息
void PushUp(int Root) {
    SegmentTree[Root].Sum = SegmentTree[Root << 1].Sum +
        ↪ SegmentTree[Root << 1 | 1].Sum;
}

// 建树, Left、Right: 当前节点区间, Root: 当前节点编号
void Build(int Left, int Right, int Root) {
    SegmentTree[Root].Left = Left;
    SegmentTree[Root].Right = Right;
    SegmentTree[Root].Lazy = 0;
```

```

SegmentTree[Root].Tag = 0;
// 叶子节点
if (Left == Right) {
    scanf("%d", &SegmentTree[Root].Sum);
    return;
}
// 左右子树
int Mid = (Left + Right) >> 1;
Build(Left, Mid, Root << 1);
Build(Mid + 1, Right, Root << 1 | 1);
// 更新
PushUp(Root);
}

// 单点更新, Pos: 修改点位置, Value: 修改值, Root: 当前节点编号
void PointUpdate(int Pos, int Value, int Root) {
    SegmentTree[Root].Sum += Value;
    if (SegmentTree[Root].Left == Pos &&
        ↪ SegmentTree[Root].Right == Pos) {
        return;
    }
    int Mid = (SegmentTree[Root].Left +
        ↪ SegmentTree[Root].Right) >> 1;
    if (Pos <= Mid) {
        PointUpdate(Pos, Value, Root << 1);
    }
    else {
        PointUpdate(Pos, Value, Root << 1 | 1);
    }
    PushUp(Root);
}

// 区间修改, Left、Right: 修改区间, Value: 修改值, Root: 当前节点
↪ 编号
void IntervalUpdate(int Left, int Right, int Value, int Root)
↪ {
    if (SegmentTree[Root].Left == Left &&
        ↪ SegmentTree[Root].Right == Right) {
        SegmentTree[Root].Lazy = 1;
        SegmentTree[Root].Tag = Value;
        SegmentTree[Root].Sum = (Right - Left + 1) * Value;
        return;
    }
}

```

```
int Mid = (SegmentTree[Root].Left +
    ↪ SegmentTree[Root].Right) >> 1;
// 下推更新
if (SegmentTree[Root].Lazy == 1) {
    SegmentTree[Root].Lazy = 0;
    IntervalUpdate(SegmentTree[Root].Left, Mid,
        ↪ SegmentTree[Root].Tag, Root << 1);
    IntervalUpdate(Mid + 1, SegmentTree[Root].Right,
        ↪ SegmentTree[Root].Tag, Root << 1 | 1);
    SegmentTree[Root].Tag = 0;
}
if (Right <= Mid) {
    IntervalUpdate(Left, Right, Value, Root << 1);
}
else if (Left > Mid) {
    IntervalUpdate(Left, Right, Value, Root << 1 | 1);
}
else {
    IntervalUpdate(Left, Mid, Value, Root << 1);
    IntervalUpdate(Mid + 1, Right, Value, Root << 1 | 1);
}
PushUp(Root);
}

// 区间查询, Left、Right: 查询区间, Root: 当前节点编号
int Query(int Left, int Right, int Root) {
    if (Left == SegmentTree[Root].Left && Right ==
        ↪ SegmentTree[Root].Right) {
        return SegmentTree[Root].Sum;
    }
    int Mid = (SegmentTree[Root].Left +
        ↪ SegmentTree[Root].Right) >> 1;
    if (Right <= Mid) {
        return Query(Left, Right, Root << 1);
    }
    else if (Left > Mid) {
        return Query(Left, Right, Root << 1 | 1);
    }
    else {
        return Query(Left, Mid, Root << 1) + Query(Mid + 1,
            ↪ Right, Root << 1 | 1);
    }
}
}
```

### 3.4 Splay

#### 3.4.1 SplayTree

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

const int maxn = "Edit";

struct SplayTree {
    // Root:Splay Tree 根节点
    int Root, Tot;
    // Son[i][0]:i 节点的左孩子, Son[i][1]:i 节点的右孩子
    int Son[maxn][2];
    // Pre[i]:i 节点的父节点
    int Pre[maxn];
    // Val[i]:i 节点的权值
    int Val[maxn];
    // Size[i]: 以 i 节点为根的 Splay Tree 的节点数 (包含自身)
    int Size[maxn];
    // Cnt[i]: 节点 i 的权值的出现次数
    int Cnt[maxn];

    void PushUp(int X) {
        Size[X] = Size[Son[X][0]] + Size[Son[X][1]] + Cnt[X];
    }

    // 判断 X 节点是其父节点的左孩子还是右孩子
    bool Self(int X) {
        return X == Son[Pre[X]][1];
    }

    void Clear(int X) {
        Son[X][0] = Son[X][1] = Pre[X] = Val[X] = Size[X] =
        ↪ Cnt[X] = 0;
    }

    // 旋转
    void Rotate(int X) {
        int Fa = Pre[X], FaFa = Pre[Fa], XJ = Self(X);
        Son[Fa][XJ] = Son[X][XJ ^ 1];
        Pre[Son[Fa][XJ]] = Pre[X];
        Son[X][XJ ^ 1] = Pre[X];
        Pre[Fa] = X;
        Pre[X] = FaFa;
    }
};
```

```

    if (FaFa) {
        Son[FaFa][Fa == Son[FaFa][1]] = X;
    }
    PushUp(Fa);
    PushUp(X);
}

// 旋转 X 节点到根节点
void Splay(int X) {
    for (int i = Pre[X]; i = Pre[X]; Rotate(X)) {
        if (Pre[i]) {
            Rotate(Self(X) == Self(i) ? i : X);
        }
    }
    Root = X;
}

// 插入数 X
void Insert(int X) {
    if (!Root) {
        Val[++Tot] = X;
        Cnt[Tot]++;
        Root = Tot;
        PushUp(Root);
        return;
    }
    int Cur = Root, F = 0;
    while (true) {
        if (Val[Cur] == X) {
            Cnt[Cur]++;
            PushUp(Cur);
            PushUp(F);
            Splay(Cur);
            break;
        }
        F = Cur;
        Cur = Son[Cur][Val[Cur] < X];
    }
    if (!Cur) {
        Val[++Tot] = X;
        Cnt[Tot]++;
        Pre[Tot] = F;
        Son[F][Val[F] < X] = Tot;
        PushUp(Tot);
        PushUp(F);
    }
}

```

```

        Splay(Tot);
        break;
    }
}

// 查询 x 的排名
int Rank(int X) {
    int Ans = 0, Cur = Root;
    while (true) {
        if (X < Val[Cur]) {
            Cur = Son[Cur][0];
        }
        else {
            Ans += Size[Son[Cur][0]];
            if (X == Val[Cur]) {
                Splay(Cur);
                return Ans + 1;
            }
            Ans += Cnt[Cur];
            Cur = Son[Cur][1];
        }
    }
}

// 查询排名为 x 的数
int Kth(int X) {
    int Cur = Root;
    while (true) {
        if (Son[Cur][0] && X <= Size[Son[Cur][0]]) {
            Cur = Son[Cur][0];
        }
        else {
            X -= Cnt[Cur] + Size[Son[Cur][0]];
            if (X <= 0) {
                return Val[Cur];
            }
            Cur = Son[Cur][1];
        }
    }
}

/*

```

```
* 在 Insert 操作时 x 已经 Splay 到根了
* 所以 x 的前驱就是 x 的左子树的最右边的节点
* 后继就是 x 的右子树的最左边的节点
*/

// 求前驱
int GetPath() {
    int Cur = Son[Root][0];
    while (Son[Cur][1]) {
        Cur = Son[Cur][1];
    }
    return Cur;
}

// 求后继
int GetNext() {
    int Cur = Son[Root][1];
    while (Son[Cur][0]) {
        Cur = Son[Cur][0];
    }
    return Cur;
}

// 删除值为 x 的节点
void Delete(int X) {
    // 将 x 旋转到根
    Rank(X);
    if (Cnt[Root] > 1) {
        Cnt[Root]--;
        PushUp(Root);
        return;
    }
    if (!Son[Root][0] && !Son[Root][1]) {
        Clear(Root);
        Root = 0;
        return;
    }
    if (!Son[Root][0]) {
        int Temp = Root;
        Root = Son[Root][1];
        Pre[Root] = 0;
        Clear(Temp);
        return;
    }
}
```



```
    if (!Son[Root][1]) {
        int Temp = Root;
        Root = Son[Root][0];
        Pre[Root] = 0;
        Clear(Temp);
        return;
    }
    int Temp = GetPath(), Old = Root;
    Splay(Temp);
    Pre[Son[Old][1]] = Temp;
    Son[Temp][1] = Son[Old][1];
    Clear(Old);
    PushUp(Root);
}
};
```

### 3.4.2 SplayTreeArray

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

const int maxn = "Edit";

// Root:Splay Tree 根节点
int Root, Tot;
// Son[i][0]:i 节点的左孩子, Son[i][1]:i 节点的右孩子
int Son[maxn][2];
// Pre[i]:i 节点的父节点
int Pre[maxn];
// Val[i]:i 节点的权值
int Val[maxn];
// Size[i]: 以 i 节点为根的 Splay Tree 的节点数 (包含自身)
int Size[maxn];
// 惰性标记数组
bool Lazy[maxn];

void PushUp(int X) {
    Size[X] = Size[Son[X][0]] + Size[Son[X][1]] + 1;
}

void PushDown(int X) {
    if (Lazy[X]) {
        std::swap(Son[X][0], Son[X][1]);
        if (Son[X][0]) {
            Lazy[Son[X][0]] ^= 1;
        }
    }
}
```

```
    }
    if (Son[X][1]) {
        Lazy[Son[X][1]] ^= 1;
    }
    Lazy[X] = 0;
}

// 判断 x 节点是其父节点的左孩子还是右孩子
bool Self(int X) {
    return Son[Pre[X]][1] == X;
}

// 旋转节点 X
void Rotate(int X) {
    int Fa = Pre[X], FaFa = Pre[Fa], XJ = Self(X);
    PushDown(Fa); PushDown(X);
    Son[Fa][XJ] = Son[X][XJ ^ 1];
    Pre[Son[Fa][XJ]] = Pre[X];
    Son[X][XJ ^ 1] = Pre[X];
    Pre[Fa] = X;
    Pre[X] = FaFa;
    if (FaFa) {
        Son[FaFa][Fa == Son[FaFa][1]] = X;
    }
    PushUp(Fa); PushUp(X);
}

// 旋转 X 节点到节点 Goal
void Splay(int X, int Goal = 0) {
    for (int Cur = Pre[X]; (Cur = Pre[X]) != Goal; Rotate(X))
        ↪ {
            PushDown(Pre[Cur]); PushDown(Cur); PushDown(X);
            if (Pre[Cur] != Goal) {
                if (Self(X) == Self(Cur)) {
                    Rotate(Cur);
                }
                else {
                    Rotate(X);
                }
            }
        }
    if (!Goal) {
        Root = X;
    }
}
```

```
    }  
}  
  
// 获取以  $R$  为根节点 Splay Tree 中的第  $K$  大个元素在 Splay Tree  
↪ 中的位置  
int Kth(int R, int K) {  
    PushDown(R);  
    int Temp = Size[Son[R][0]] + 1;  
    if (Temp == K) {  
        return R;  
    }  
    if (Temp > K) {  
        return Kth(Son[R][0], K);  
    }  
    else {  
        return Kth(Son[R][1], K - Temp);  
    }  
}  
  
// 获取 Splay Tree 中以  $X$  为根节点子树的最小值位置  
int GetMin(int X) {  
    PushDown(X);  
    while (Son[X][0]) {  
        X = Son[X][0];  
        PushDown(X);  
    }  
    return X;  
}  
  
// 获取 Splay Tree 中以  $X$  为根节点子树的最大值位置  
int GetMax(int X) {  
    PushDown(X);  
    while (Son[X][1]) {  
        X = Son[X][1];  
        PushDown(X);  
    }  
    return X;  
}  
  
// 求节点  $X$  的前驱节点  
int GetPath(int X) {  
    Splay(X, Root);  
    int Cur = Son[Root][0];
```

```
    while (Son[Cur][1]) {
        Cur = Son[Cur][1];
    }
    return Cur;
}

// 求节点 Y 的后继节点
int GetNext(int X) {
    Splay(X, Root);
    int Cur = Son[Root][1];
    while (Son[Cur][0]) {
        Cur = Son[Cur][0];
    }
    return Cur;
}

// 翻转 Splay Tree 中 Left~Right 区间
void Reverse(int Left, int Right) {
    int X = Kth(Root, Left), Y = Kth(Root, Right);
    Splay(X, 0);
    Splay(Y, X);
    Lazy[Son[Y][0]] ^= 1;
}

// 建立 Splay Tree
void Build(int Left, int Right, int Cur) {
    if (Left > Right) {
        return;
    }
    int Mid = (Left + Right) >> 1;
    Build(Left, Mid - 1, Mid);
    Build(Mid + 1, Right, Mid);
    Pre[Mid] = Cur;
    Val[Mid] = Mid - 1;
    Lazy[Mid] = 0;
    PushUp(Mid);
    if (Mid < Cur) {
        Son[Cur][0] = Mid;
    }
    else {
        Son[Cur][1] = Mid;
    }
}
```

```
// 输出 Splay Tree
void Print(int Cur) {
    PushDown(Cur);
    if (Son[Cur][0]) {
        Print(Son[Cur][0]);
    }
    // 哨兵节点判断
    if (Val[Cur] != -INF && Val[Cur] != INF) {
        printf("%d ", Val[Cur]);
    }
    if (Val[Son[Cur][1]]) {
        Print(Son[Cur][1]);
    }
}
```

### 3.5 TrieTree

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

const int maxn = "Edit";

struct Trie {
    // Trie Tree 节点
    int Son[maxn][26];
    // Trie Tree 节点数量
    int Tot;

    // 字符串数量统计数组
    int Cnt[maxn];

    // Trie Tree 初始化
    void TrieInit() {
        Tot = 0;
        memset(Cnt, 0, sizeof(Cnt));
        memset(Son, 0, sizeof(Son));
    }

    // 计算字母下标
    int Pos(char X) {
        return X - 'a';
    }

    // 向 Trie Tree 中插入字符串 Str
```

```
void Insert(string Str) {
    int Cur = 0, Len = int(Str.length());
    for (int i = 0; i < Len; ++i) {
        int Index = Pos(Str[i]);
        if (!Son[Cur][Index]) {
            Son[Cur][Index] = ++Tot;
        }
        Cur = Son[Cur][Index];

        Cnt[Cur]++;
    }
}

// 查找字符串 Str, 存在返回 true, 不存在返回 false
bool Find(string Str) {
    int Cur = 0, Len = int(Str.length());
    for (int i = 0; i < Len; ++i) {
        int Index = Pos(Str[i]);
        if (!Son[Cur][Index]) {
            return false;
        }
        Cur = Son[Cur][Index];
    }
    return true;
}

// 查询字典树中以 Str 为前缀的字符串数量
int PathCnt(string Str) {
    int Cur = 0, Len = int(Str.length());
    for (int i = 0; i < Len; ++i) {
        int Index = Pos(Str[i]);
        if (!Son[Cur][Index]) {
            return 0;
        }
        Cur = Son[Cur][Index];
    }
    return Cnt[Cur];
}
};
```

## 4 GraphTheory

### 4.1 LCA

#### 4.1.1 DFS+ST

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

const int maxn = "Edit";

// 链式前向星存图
struct Edge {
    int V, Weight, Next;
};

Edge edges[maxn << 1];
int Head[maxn];
int Tot;

void Init() {
    Tot = 0;
    memset(Head, -1, sizeof(Head));
}

void AddEdge(int U, int V, int Weight) {
    edges[Tot] = Edge {V, Weight, Head[U]};
    Head[U] = Tot++;
}

struct LCAOnline {
    // 节点深度
    int Rmq[maxn << 1];
    // 深搜遍历顺序
    int Vertex[maxn << 1];
    // 节点在深搜中第一次出现的位置
    int First[maxn];
    // 记录父节点
    int Parent[maxn];
    // 记录与根节点距离
    int Dis[maxn];
    // 遍历节点数量
    int LCATot;

    // 最小值对应下标
    int Dp[maxn << 1][20];
};
```

```
// RMQ 初始化
void Work(int N) {
    for (int i = 1; i <= N; ++i) {
        Dp[i][0] = i;
    }
    for (int j = 1; (1 << j) <= N; ++j) {
        for (int i = 1; i + (1 << j) - 1 <= N; ++i) {
            Dp[i][j] = Rmq[Dp[i][j - 1]] < Rmq[Dp[i + (1
↪ << (j - 1))] [j - 1]] ? Dp[i][j - 1] : Dp[i
↪ + (1 << (j - 1))] [j - 1];
        }
    }
}

// 深搜
void Dfs(int Cur, int Pre, int Depth) {
    Vertex[++LCATot] = Cur;
    First[Cur] = LCATot;
    Rmq[LCATot] = Depth;
    Parent[Cur] = Pre;
    for (int i = Head[Cur]; ~i; i = edges[i].Next) {
        if (edges[i].V == Pre) {
            continue;
        }
        Dis[edges[i].V] = Dis[Cur] + edges[i].Weight;
        Dfs(edges[i].V, Cur, Depth + 1);
        Vertex[++LCATot] = Cur;
        Rmq[LCATot] = Depth;
    }
}

// RMQ 查询
int Query(int Left, int Right) {
    if (Left > Right) {
        swap(Left, Right);
    }
    int Len = int(log2(Right - Left + 1));
    return Rmq[Dp[Left][Len]] <= Rmq[Dp[Right - (1 << Len)
↪ + 1][Len]] ? Dp[Left][Len] : Dp[Right - (1 << Len)
↪ + 1][Len];
}

// LCA 初始化
```



```
void Init(int Root, int NodeNum) {
    memset(Dis, 0, sizeof(Dis));
    LCATot = 0;
    Dfs(Root, 0, 0);
    Parent[1] = 0;
    Work(2 * NodeNum - 1);
}

// 查询节点 U V 的距离
int GetDis(int U, int V) {
    return Dis[U] + Dis[V] - 2 * Dis[LCA(U, V)];
}

// 查询节点 U V 的最近公共祖先 (LCA)
int LCA(int U, int V) {
    return Vertex[Query(First[U], First[V])];
}
}LCA;
```

#### 4.1.2 Tarjan

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

const int maxn = "Edit";

// 树边
struct Edge {
    int V, Next;
};

// 询问
struct Query {
    int Q, Next;
    int Index;
};

// 并查集数组
int Pre[maxn << 2];
// 树边
Edge edges[maxn << 2];
int Head[maxn];
int Tot;
// 询问
Query querys[maxn << 2];
```

```
int QHead[maxn];
int QTot;
// 访问标记
int Vis[maxn];
int Ancestor[maxn];
// 结果
int Answer[maxn];

// 并查集查找
int Find(int X) {
    int R = X;
    while (Pre[R] != -1) {
        R = Pre[R];
    }
    return R;
}

// 并查集合并
void Join(int U, int V) {
    int RU = Find(U);
    int RV = Find(V);
    if (RU != RV) {
        Pre[RU] = RV;
    }
}

// 添加树边
void AddEdge(int U, int V) {
    edges[Tot] = Edge {V, Head[U]};
    Head[U] = Tot++;
}

// 添加询问
void AddQuery(int U, int V, int Index) {
    queries[QTot] = Query {V, QHead[U], Index};
    QHead[U] = QTot++;
    queries[QTot] = Query {U, QHead[V], Index};
    QHead[V] = QTot++;
}

// 初始化
void Init() {
    Tot = 0;
    memset(Head, -1, sizeof(Head));
}
```

```
    QTot = 0;
    memset(QHead, -1, sizeof(QHead));
    memset(Vis, false, sizeof(Vis));
    memset(Pre, -1, sizeof(Pre));
    memset(Ancestor, 0, sizeof(Ancestor));
}

// LCA 离线 Tarjan 算法
void Tarjan(int Node) {
    Ancestor[Node] = Node;
    Vis[Node] = true;
    for (int i = Head[Node]; i != -1; i = edges[i].Next) {
        if (Vis[edges[i].V]) {
            continue;
        }
        Tarjan(edges[i].V);
        Join(Node, edges[i].V);
        Ancestor[Find(Node)] = Node;
    }
    for (int i = QHead[Node]; i != -1; i = querys[i].Next) {
        if (Vis[querys[i].Q]) {
            Answer[querys[i].Index] =
                Ancestor[Find(querys[i].Q)];
        }
    }
}
```

## 4.2 MinimumSpanningTree

### 4.2.1 Kruskal

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

const int maxn = "Edit";

struct Edge {
    int U, V, Dis;
};

// N: 顶点数, E: 边数, Pre 并查集
int N, E, Pre[maxn];
// edges: 边
Edge edges[maxn];
```

```
void Init() {
    // 并查集初始化
    for (int i = 0; i <= N; ++i) {
        Pre[i] = i;
    }
}

// 并查集查询
int Find(int X) {
    int R = X;
    while (Pre[R] != R) {
        R = Pre[R];
    }
    int I = X, J;
    while (I != R) {
        J = Pre[I];
        Pre[I] = R;
        I = J;
    }
    return R;
}

// 并查集合并
void Join(int X, int Y) {
    int XX = Find(X);
    int YY = Find(Y);
    if (XX != YY) {
        Pre[XX] = YY;
    }
}

// Kruskal 算法
int Kruskal() {
    // 贪心排序
    std::sort(edges + 1, edges + E + 1);
    Init();
    int Res = 0;
    // 选边计算
    for (int i = 1; i <= E; ++i) {
        Edge Temp = edges[i];
        if (Find(Temp.U) != Find(Temp.V)) {
            Join(Temp.U, Temp.V);
            Res += Temp.Dis;
        }
    }
}
```

```
    }  
    return Res;  
}
```

#### 4.2.2 Prim

```
#include <bits/stdc++.h>  
  
const int INF = "Edit";  
const int maxn = "Edit";  
  
struct Edge {  
    // V: 连接点, Dis: 边权  
    int V, Dis;  
};  
  
// N: 顶点数, E: 边数  
int N, E;  
// 松弛更新权值数组  
int Dis[maxn];  
// 访问标记数组  
int Vis[maxn];  
// 邻接表  
std::vector<Edge> Adj[maxn];  
  
// 建图加边, U, V: 顶点, Weight: 权值  
void AddEdge(int U, int V, int Weight) {  
    Adj[U].push_back(Edge (V, Weight));  
    // 无向图反向建边  
    Adj[V].push_back(Edge (U, Weight));  
}  
  
// Prim 算法  
int Prim(int Start) {  
    memset(Dis, INF, sizeof(Dis));  
    memset(Vis, 0, sizeof(Vis));  
    Dis[Start] = 0;  
    int Res = 0;  
    for (int i = 1; i <= N; ++i) {  
        // 选择距已生成树权值最小的顶点  
        int U = -1, Min = INF;  
        for (int j = 1; j <= N; ++j) {  
            if (!Vis[j] && Dis[j] < Min) {  
                U = j;  
            }  
        }  
        if (U == -1) break;  
        Vis[U] = 1;  
        Res += Dis[U];  
        for (int j = 0; j < Adj[U].size(); ++j) {  
            int V = Adj[U][j].V;  
            int w = Adj[U][j].Dis;  
            if (!Vis[V] && Dis[V] > Dis[U] + w) {  
                Dis[V] = Dis[U] + w;  
            }  
        }  
    }  
    return Res;  
}
```

```
        Min = Dis[j];
    }
}
// 更新、标记
Vis[U] = 1;
Res += Min;
// 松弛
for (int j = 0; j < int(Adj[U].size()); ++j) {
    int V = Adj[U][j].V;
    if (!Vis[V] && Adj[U][j].Dis < Dis[V]) {
        Dis[V] = Adj[U][j].Dis;
    }
}
}
// 返回结果
return Res;
}
```

## 4.3 NetworkFlow

### 4.3.1 Dinic

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

const int INF = "Edit";
const int maxn = "Edit";

// 边
struct Edge {
    // V: 连接点, Weight: 权值, Next: 上一条边的编号
    int V, Weight, Next;
};

// 边, 一定要开到足够大
Edge edges[maxn << 1];
// Head[i] 为点 i 上最后一条边的编号
int Head[maxn];
// 增加边时更新编号
int Tot;
// N: 顶点数, E: 边数
int N, E;
// Bfs 分层深度
int Depth[maxn];
// 当前弧优化
```

```
int Current[maxn];

// 链式向前星初始化
void Init() {
    Tot = 0;
    memset(Head, -1, sizeof(Head));
}

// 添加一条由  $U$  至  $V$  权值为  $Weight$  的边
void AddEdge(int U, int V, int Weight, int ReverseWeight = 0)
↪ {
    edges[Tot] = Edge (V, Weight, Head[U]);
    Head[U] = Tot++;
    // 反向建边
    edges[Tot] = Edge (U, ReverseWeight, Head[V]);
    Head[V] = Tot++;
}

// Bfs 搜索分层图,  $Start$ : 起点,  $End$ : 终点
bool Bfs(int Start, int End) {
    memset(Depth, -1, sizeof(Depth));
    std::queue<int> Que;
    Depth[Start] = 0;
    Que.push(Start);
    while (!Que.empty()) {
        int Cur = Que.front();
        Que.pop();
        for (int i = Head[Cur]; ~i; i = edges[i].Next) {
            if (Depth[edges[i].V] == -1 && edges[i].Weight >
↪ 0) {
                Depth[edges[i].V] = Depth[Cur] + 1;
                Que.push(edges[i].V);
            }
        }
    }
    return Depth[End] != -1;
}

// Dfs 搜索增广路径,  $Cur$ : 当前搜索顶点,  $End$ : 终点,  $NowFlow$ : 当前
↪ 最大流
int Dfs(int Cur, int End, int NowFlow) {
    // 搜索到终点或者可用当前最大流为 0 返回
    if (Cur == End || NowFlow == 0) {
```

```

        return NowFlow;
    }
    // UsableFlow: 可用流量, 当达到 NowFlow 时不可再增加,
    // ↪ FindFlow: 递归深搜到的最大流
    int UsableFlow = 0, FindFlow;
    // &i=Current[Cur] 为当前弧优化, 每次更新 Current[Cur]
    for (int &i = Current[Cur]; ~i; i = edges[i].Next) {
        if (edges[i].Weight > 0 && Depth[edges[i].V] ==
            // ↪ Depth[Cur] + 1) {
            FindFlow = Dfs(edges[i].V, End, std::min(NowFlow -
                // ↪ UsableFlow, edges[i].Weight));
            if (FindFlow > 0) {
                edges[i].Weight -= FindFlow;
                // 反边
                edges[i ^ 1].Weight += FindFlow;
                UsableFlow += FindFlow;
                if (UsableFlow == NowFlow) {
                    return NowFlow;
                }
            }
        }
    }
    // 炸点优化
    if (!UsableFlow) {
        Depth[Cur] = -2;
    }
    return UsableFlow;
}

// Dinic 算法, Start: 起点, End: 终点
int Dinic(int Start, int End) {
    int MaxFlow = 0;
    while (Bfs(Start, End)) {
        // 当前弧优化
        for (int i = 1; i <= N; ++i) {
            Current[i] = Head[i];
        }
        MaxFlow += Dfs(Start, End, INF);
    }
    // 返回结果
    return MaxFlow;
}

```



### 4.3.2 FordFulkerson

```
#include <bits/stdc++.h>
// 正无穷
const int INF = "Edit";
const int maxn = "Edit";

// N: 顶点数, E: 边数
int N, E;
// 访问标记数组
bool Vis[maxn];
// 邻接矩阵
int Adj[maxn][maxn];

// Dfs 搜索增广路径, Vertex: 当前搜索顶点, End: 搜索终点,
// ↪ NowFlow: 当前最大流量
int Dfs(int Vertex, int End, int NowFlow) {
    // 搜索到终点结束
    if (Vertex == End) {
        return NowFlow;
    }
    // 标记访问过的顶点
    Vis[Vertex] = true;
    // 枚举寻找顶点
    for (int i = 1; i <= N; ++i) {
        if (!Vis[i] && Adj[Vertex][i]) {
            int FindFlow = Dfs(i, End, NowFlow <
                ↪ Adj[Vertex][i] ? NowFlow : Adj[Vertex][i]);
            if (!FindFlow) {
                continue;
            }
            // 找到增广路径后更新邻接矩阵残留网
            Adj[Vertex][i] -= FindFlow;
            Adj[i][Vertex] += FindFlow;
            // 返回搜索结果
            return FindFlow;
        }
    }
    // 未找到增广路径, 搜索失败
    return false;
}

// Ford-Fulkerson 算法, Start: 起点, End: 终点
int FordFulkerson(int Start, int End) {
```

```
// MaxFlow: 最大流, Flow: 搜索到的增广路径最大流
int MaxFlow = 0, Flow = 0;
memset(Vis, false, sizeof(Vis));
// 搜索增广路径
while (Flow = Dfs(Start, End, INF)) {
    MaxFlow += Flow;
    memset(Vis, false, sizeof(Vis));
}
// 返回结果
return MaxFlow;
}
```

### 4.3.3 MinCostMaxFlow

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

const int INF = "Edit";
const int maxn = "Edit";

// 边
struct Edge {
    // V: 连接点, Flow: 流量, Cost: 费用
    int V, Cap, Cost, Flow, Next;
};

// N: 顶点数, E: 边数
int N, E;
int Head[maxn];
// 前驱记录数组
int Path[maxn];
int Dis[maxn];
// 访问标记数组
bool Vis[maxn];
int Tot;
// 链式前向星
Edge edges[maxn];

// 链式前向星初始化
void Init() {
    Tot = 0;
    memset(Head, -1, sizeof(Head));
}

// 建图加边, U V 之间建立一条费用为 Cost 的边
```

```

void AddEdge(int U, int V, int Cap, int Cost) {
    edges[Tot] = Edge {V, Cap, Cost, 0, Head[U]};
    Head[U] = Tot++;
    edges[Tot] = Edge {U, 0, -Cost, 0, Head[V]};
    Head[V] = Tot++;
}

// SPFA 算法, Start: 起点, End: 终点
bool SPFA(int Start, int End) {
    memset(Dis, INF, sizeof(Dis));
    memset(Vis, false, sizeof(Vis));
    memset(Path, -1, sizeof(Path));
    Dis[Start] = 0;
    Vis[Start] = true;
    std::queue<int> Que;
    while (!Que.empty()) {
        Que.pop();
    }
    Que.push(Start);
    while (!Que.empty()) {
        int U = Que.front();
        Que.pop();
        Vis[U] = false;
        for (int i = Head[U]; ~i; i = edges[i].Next) {
            int V = edges[i].V;
            if (edges[i].Cap > edges[i].Flow && Dis[V] >
                Dis[U] + edges[i].Cost) {
                Dis[V] = Dis[U] + edges[i].Cost;
                Path[V] = i;
                if (!Vis[V]) {
                    Vis[V] = true;
                    Que.push(V);
                }
            }
        }
    }
    return Path[End] != -1;
}

// 最小费用最大流, Start: 起点, End: 终点, Cost: 最小费用
int MinCostMaxFlow(int Start, int End, int &MinCost) {
    int MaxFlow = 0;
    MinCost = 0;
    while (SPFA(Start, End)) {

```

```
    int Min = INF;
    for (int i = Path[End]; ~i; i = Path[edges[i ^ 1].V])
        ↪ {
            if (edges[i].Cap - edges[i].Flow < Min) {
                Min = edges[i].Cap - edges[i].Flow;
            }
        }
    for (int i = Path[End]; ~i; i = Path[edges[i ^ 1].V])
        ↪ {
            edges[i].Flow += Min;
            edges[i ^ 1].Flow -= Min;
            MinCost += edges[i].Cost * Min;
        }
    MaxFlow += Min;
}
// 返回最大流
return MaxFlow;
}
```

## 4.4 ShortestPath

### 4.4.1 BellmanFord

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

const int INF = "Edit";
const int maxn = "Edit";

struct Edge {
    // U, V: 顶点, Dis: 边权
    int U, V;
    int Dis;
};
// 松弛更新数组
int Dis[maxn];
// 边
std::vector<Edge> edges;

// Bellman_Ford 算法判断是否存在负环回路
bool BellmanFord(int Start, int N) {
    memset(Dis, INF, sizeof(Dis));
    Dis[Start] = 0;
    // 最多做 N-1 次
    for (int i = 1; i < N; ++i) {
```

```
    bool flag = false;
    for (int j = 0; j < int(edges.size()); ++j) {
        if (Dis[edges[j].V] > Dis[edges[j].U] +
            ↪ edges[j].Dis) {
            Dis[edges[j].V] = Dis[edges[j].U] +
            ↪ edges[j].Dis;
            flag = true;
        }
    }
    // 没有负环回路
    if (!flag) {
        return true;
    }
}
// 有负环回路
for (int j = 0; j < int(edges.size()); ++j) {
    if (Dis[edges[j].V] > Dis[edges[j].U] + edges[j].Dis)
        ↪ {
            return false;
        }
}
// 没有负环回路
return true;
}
```

#### 4.4.2 Dijkstra

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

const int maxn = "Edit";
const int INF = "Edit";

// 边
struct Edge {
    // V: 连接点, Weight: 权值, Next: 上一条边的编号
    int V, Weight, Next;
};

// 边, 一定要开到足够大
Edge edges[maxn << 1];
// Head[i] 为点 i 上最后一条边的编号
int Head[maxn];
// 增加边时更新编号
int Tot;
```

```
// 松弛更新数组, 最短路
int Dis[maxn];

// 链式前向星初始化
void Init() {
    Tot = 0;
    memset(Head, -1, sizeof(Head));
}

// 添加一条 U 至 V 权值为 Weight 的边
void AddEdge(int U, int V, int Weight) {
    edges[Tot] = Edge (V, Weight, Head[U]);
    Head[U] = Tot++;
}

// 最短路优化堆排序规则
struct Cmp {
    bool operator() (const int &A, const int &B) {
        return Dis[A] > Dis[B];
    }
};

// N: 顶点数, E: 边数
int N, E;

// Dijkstra 算法, Start: 起点
void Dijkstra(int Start) {
    std::priority_queue<int, std::vector<int>, Cmp> Que;
    memset(Dis, INF, sizeof(Dis));
    Dis[Start] = 0;
    Que.push(Start);
    while (!Que.empty()) {
        int U = Que.top(); Que.pop();
        for (int i = Head[U]; ~i; i = edges[i].Next) {
            if (Dis[edges[i].V] > Dis[U] + edges[i].Weight) {
                Dis[edges[i].V] = Dis[U] + edges[i].Weight;
                Que.push(edges[i].V);
            }
        }
    }
}
```

### 4.4.3 Floyd

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

const int maxn = "Edit";

// N: 顶点数
int N;
// Dis[i][j] 为 i 点到 j 点的最短路
int Dis[maxn][maxn];

// Floyd 算法
void Floyd() {
    for (int k = 1; k <= N; ++k) {
        for (int i = 1; i <= N; ++i) {
            for (int j = 1; j <= N; ++j) {
                Dis[i][j] = std::min(Dis[i][j], Dis[i][k] +
                    ↪ Dis[k][j]);
            }
        }
    }
}
```

### 4.4.4 SPFA

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

const int INF = "Edit";
const int maxn = "Edit";

// 边
struct Edge {
    // V: 连接点, Dis: 边权
    int V, Dis;
};

// N: 顶点数, E: 边数
int N, E;
// 访问标记数组
bool Vis[maxn];
// 每个点的入队列次数
int Cnt[maxn];
// 最短路数组
int Dis[maxn];
```

```
// 邻接表
std::vector<Edge> Adj[maxn];

// 建图加边,  $u$   $v$  之间权值为  $Weight$  的边
void AddEdge (int U, int V, int Weight) {
    Adj[U].push_back(Edge (V, Weight));
    // 无向图建立反向边
    Adj[V].push_back(Edge (U, Weight));
}

// SPFA 算法,  $Start$ : 起点
bool SPFA(int Start) {
    memset(Vis, false, sizeof(Vis));
    memset(Dis, INF, sizeof(Dis));
    memset(Cnt, 0, sizeof(Cnt));
    Vis[Start] = true;
    Dis[Start] = 0;
    Cnt[Start] = 1;
    std::queue<int> Que;
    while (!Que.empty()) {
        Que.pop();
    }
    Que.push(Start);
    while (!Que.empty()) {
        int U = Que.front();
        Que.pop();
        Vis[U] = false;
        for (int i = 0; i < int(Adj[U].size()); ++i) {
            int V = Adj[U][i].V;
            if (Dis[V] > Dis[U] + Adj[U][i].Dis) {
                Dis[V] = Dis[U] + Adj[U][i].Dis;
                if (!Vis[V]) {
                    Vis[V] = true;
                    Que.push(V);
                    //  $Cnt[i]$  为  $i$  顶点入队列次数, 用来判定是否
                    // 存在负环回路
                    if (++Cnt[V] > N) {
                        return false;
                    }
                }
            }
        }
    }
}
```



```
    return true;  
}
```

## 5 DynamicProgramming

### 5.1 Contour

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

const int maxn = "Edit";

int Dp[2][1 << maxn];

void Update(int Cur, int A, int B) {
    if (B & (1 << M)) {
        Dp[Cur][B ^ (1 << M)] = Dp[Cur][B ^ (1 << M)] + Dp[Cur
        ↪ ^ 1][A];
    }
}

// 轮廓线 Dp(1*2 在 N*M 图上摆放数)
int Contour(int N, int M) {
    memset(Dp, 0, sizeof(Dp));
    int Cur = 0;
    Dp[Cur][(1 << M) - 1] = 1;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < M; ++j) {
            Cur ^= 1;
            memset(Dp[Cur], 0, sizeof(Dp[Cur]));
            for (int k = 0; k < (1 << M); ++k) {
                Update(Cur, k, k << 1);
                if (i && !(k & (1 << (M - 1)))) {
                    Update(Cur, k, (k << 1) ^ (1 << M) ^ 1);
                }
                if (j && !(k & 1)) {
                    Update(Cur, k, (k << 1) ^ 3);
                }
            }
        }
    }
    return Dp[Cur][(1 << M) - 1];
}
```

### 5.2 Digit

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

const int maxn = "Edit";
```

```
long long Digit[25];
long long Dp[25][maxn];

// Site: 数位, Statu: 状态, Pre: 前导零, Limit: 数位上界
long long Dfs(long long Site, long long Statu, bool Pre, bool
↪ Limit) {
    if (Site == 0) {
        return ?;
    }
    if (!Limit && ~Dp[Site][Statu]) {
        return Dp[Site][Statu];
    }
    long long Max = Limie ? Digit[Site] : 9;
    long long Ans = 0;
    for (int i = 0; i <= Max; ++i) {
        long long NowStatu = /* 状态转移 */;
        if (NowStatu?) {
            Ans += Dfs(Site - 1, NowStatu, Pre && i == 0,
↪ Limit && i == Max);
        }
    }
    if (!Limit) {
        Dp[Site][Statu] = Ans;
    }
    return Ans;
}

long long Cal(long long X) {
    // 数位分解
    long long Len = 0;
    while (X) {
        Digit[++Len] = X % 10;
        X /= 10;
    }
    return Dfs(Len, 0, true, true);
}
```

### 5.3 LCS

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

const int maxn = "Edit";
```

```
// Dp[i][j]:Str1[1]~Str1[i] 和 Str2[1]~Str2[j] 对应的公共子序列
↪ 长度
int Dp[maxn][maxn];

// 最长公共子序列 (LCS)
void LCS(std::string Str1, std::string Str2) {
    for (int i = 0; i < int(Str1.length()); ++i) {
        for (int j = 0; j < int(Str2.length()); ++j) {
            if (Str1[i] == Str2[j]) {
                Dp[i + 1][j + 1] = Dp[i][j] + 1;
            }
            else {
                Dp[i + 1][j + 1] = std::max(Dp[i][j + 1], Dp[i
↪ + 1][j]);
            }
        }
    }
}
```

## 5.4 LIS

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

// 最长不上降子序列 (LIS), Num: 序列
int LIS(std::vector<int> &Num) {
    int Ans = 1;
    // Last[i] 为长度为 i 的不上降子序列末尾元素的最小值
    std::vector<int> Last(int(Num.size()) + 1, 0);
    Last[1] = Num[1];
    for (int i = 2; i <= int(Num.size()); ++i) {
        if (Num[i] >= Last[Ans]) {
            Last[++Ans] = Num[i];
        }
        else {
            int Index = std::upper_bound(Last.begin() + 1,
↪ Last.end(), Num[i]) - Last.begin();
            Last[Index] = Num[i];
        }
    }
    // 返回结果
    return Ans;
}
```

## 5.5 Pack

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

const int maxn = "Edit";

int Dp[maxn];
// NValue: 背包容量, NKind: 总物品数
int NValue, NKind;

// 01 背包, 代价为 Cost, 获得的价值为 Weight
void ZeroOnePack(int Cost, int Weight) {
    for (int i = NValue; i >= Cost; --i) {
        Dp[i] = std::max(Dp[i], Dp[i - Cost] + Weight);
    }
}

// 完全背包, 代价为 Cost, 获得的价值为 Weight
void CompletePack(int Cost, int Weight) {
    for (int i = Cost; i <= NValue; ++i) {
        Dp[i] = std::max(Dp[i], Dp[i - Cost] + Weight);
    }
}

// 多重背包, 代价为 Cost, 获得的价值为 Weight, 数量为 Amount
void MultiplePack(int Cost, int Weight, int Amount) {
    if (Cost * Amount >= NValue) {
        CompletePack(Cost, Weight);
    }
    else {
        int k = 1;
        while (k < Amount) {
            ZeroOnePack(k * Cost, k * Weight);
            Amount -= k;
            k <<= 1;
        }
        ZeroOnePack(Amount * Cost, Amount * Weight);
    }
}
```

## 6 ComputationalGeometry

### 6.1 Plane

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

const double eps = "Edit";

int Sgn(double X) {
    if (fabs(X) < eps) {
        return 0;
    }
    return X < 0 ? -1 : 1;
}

// 点
struct Point {
    // X: 横坐标, Y: 纵坐标
    double X, Y;

    Point() {}
    Point(double _X, double _Y) {
        X = _X;
        Y = _Y;
    }

    void input() {
        scanf("%lf%lf", &X, &Y);
    }

    // 减法
    Point operator - (const Point &B) const {
        return Point (X - B.X, Y - B.Y);
    }

    // 加法
    Point operator + (const Point &B) const {
        return Point (X + B.X, Y + B.Y);
    }

    // 点积
    double operator * (const Point &B) const {
        return X * B.X + Y * B.Y;
    }
}
```

doub

```
    // 叉积
    double operator ^ (const Point &B) const {
        return X * B.Y - Y * B.X;
    }

    // 两点间距离
    double Dis(Point &B) {
        return sqrt((B - *this) * (B - *this));
    }
};

// 线
struct Line {
    Point S, T;

    Line() {}
    Line(Point _S, Point _T) {
        S = _S;
        T = _T;
    }

    void Input() {
        S.Input();
        T.Input();
    }

    // 向量叉积
    double operator ^ (const Line &B) const {
        return (T - S) ^ (B.T - B.S);
    }

    // 判断是否平行
    bool IsParallel(Line &B) {
        return Sgn((S - T) ^ (B.S - B.T)) == 0;
    }

    // 求交点
    Point operator & (const Line &B) const {
        double Temp = ((S - B.S) ^ (B.S - B.T)) / ((S - T) ^
            ↪ (B.S - B.T));
        return Point(S.X + (T.X - S.X) * Temp, S.Y + (T.Y -
            ↪ S.Y) * Temp);
    }
};
```

```

    }
};

// 判断线段 A、B 是否相交
bool IsIntersect(Line A, Line B) {
    return
        max(A.S.X, A.T.X) >= min(B.S.X, B.T.X) &&
        max(B.S.X, B.T.X) >= min(A.S.X, A.T.X) &&
        max(A.S.Y, A.T.Y) >= min(B.S.Y, B.T.Y) &&
        max(B.S.Y, B.T.Y) >= min(A.S.Y, A.T.Y) &&
        Sgn((B.S - A.T) ^ (A.S - A.T)) * Sgn((B.T - A.T) ^
        ↪ (A.S - A.T)) <= 0 &&
        Sgn((A.S - B.T) ^ (B.S - B.T)) * Sgn((A.T - B.T) ^
        ↪ (B.S - B.T)) <= 0;
}

// 判断线段 A 所在直线与线段 B 是否相交
bool IsIntersect(Line A, Line B) {
    return Sgn((B.S - A.T) ^ (A.S - A.T)) * Sgn((B.T - A.T) ^
    ↪ (A.S - A.T)) <= 0;
}

// 判断直线 A、B 是否相交
bool IsIntersect(Line A, Line B) {
    return !Parallel(A, B) || (Parallel(A, B) && !(Sgn((A.S -
    ↪ B.S) ^ (B.T - B.S)) == 0));
}

// 判断 N 个点 (下标 1~N-1) 能否组成凸包
bool IsConvexHull(Point points[], int N) {
    for (int i = 0; i < N; ++i) {
        if (Sgn((points[(i + 1) % N] - points[i]) ^ (points[(i
        ↪ + 2) % N] - points[(i + 1) % N])) < 0) {
            return false;
        }
    }
    return true;
}

// 凸包, points: 所有点, 返回凸包总长度
double ConvexHull(std::vector<Point> points) {
    int N = int(points.size());
    // 特判点数小于等于 2 的情况

```



```
if (N == 1) {
    return 0;
}
else if (N == 2) {
    return Distance(points[0], points[1]);
}
// 查找最左下角的基准点
int Basic = 0;
for (int i = 0; i < N; ++i) {
    if (points[i].Y > points[Basic].Y ||
        (points[i].Y == points[Basic].Y && points[i].X <
         points[Basic].X)) {
        Basic = i;
    }
}
std::swap(points[0], points[Basic]);
// 对其它点进行极角排序
std::sort(points.begin() + 1, points.end(), [&] (const
    ↪ Point &A, const Point &B) {
    double Temp = (A - points[0]) ^ (B - points[0]);
    if (Temp > 0) {
        return true;
    }
    else if (!Temp && Distance(A, points[0]) < Distance(B,
    ↪ points[0])) {
        return true;
    }
    return false;
});
// 凸包选点
std::vector<Point> Stack;
Stack.push_back(points[0]);
for (int i = 2; i < N; ++i) {
    while (Stack.size() >= 2 && ((Stack.back() -
    ↪ Stack[int(Stack.size()) - 2]) ^ (points[i] -
    ↪ Stack[int(Stack.size()) - 2])) <= 0) {
        Stack.pop_back();
    }
}
Stack.push_back(points[0]);
// 计算总长
double Ans = 0;
for (int i = 1; i < int(Stack.size()); ++i) {
    Ans += Distance(Stack[i], Stack[i - 1]);
}
```

```

    }
    // 返回结果
    return Ans;
}

// 半平面, 表示 S->T 逆时针 (左侧) 的半平面
struct HalfPlane:public Line {
    double Angle;

    HalfPlane() {}
    HalfPlane(Point _S, Point _T) {
        S = _S;
        T = _T;
    }
    HalfPlane(Line ST) {
        S = ST.S;
        T = ST.T;
    }

    void CalAngle() {
        Angle = atan2(T.Y - S.Y, T.X - S.X);
    }

    bool operator < (const HalfPlane &B) const {
        if (fabs(Angle - B.Angle) > eps) {
            return Angle < B.Angle;
        }
        return ((S - B.S) ^ (B.T - B.S)) < 0;
    }
};

// 半平面交
struct HPI {
    // 半平面数量
    int Tot;
    // 半平面
    HalfPlane halfplanes[maxn];
    // 半平面交双向队列
    HalfPlane Deque[maxn];
    // 点队列
    Point points[maxn];
    // 半平面交内核
    Point Res[maxn];
    // 双向队列首尾指针

```

```
int Front, Tail;

// 添加半平面
void Push(HalfPlane X) {
    halfplanes[Tot++] = X;
}

// 半平面去重
void Unique() {
    int Cnt = 1;
    for (int i = 1; i < Tot; ++i) {
        if (fabs(halfplanes[i].Angle - halfplanes[i - 1].Angle) > eps) {
            halfplanes[Cnt++] = halfplanes[i];
        }
    }
    Tot = Cnt;
}

// 判断半平面交是否有内核
bool HalfPlaneInsert() {
    for (int i = 0; i < Tot; ++i) {
        halfplanes[i].CalAngle();
    }
    sort(halfplanes, halfplanes + Tot);
    Unique();
    Deque[Front = 0] = halfplanes[0];
    Deque[Tail = 1] = halfplanes[1];
    for (int i = 2; i < Tot; ++i) {
        if (fabs((Deque[Tail].T - Deque[Tail].S) ^
            ↪ (Deque[Tail - 1].T - Deque[Tail - 1].S)) < eps
            ↪ || fabs((Deque[Front].T - Deque[Front].S) ^
            ↪ (Deque[Front + 1].T - Deque[Front + 1].S)) <
            ↪ eps) {
            return false;
        }
        while (Front < Tail && (((Deque[Tail] & Deque[Tail - 1]) - halfplanes[i].S) ^ (halfplanes[i].T - halfplanes[i].S)) > eps) {
            Tail--;
        }
        while (Front < Tail && (((Deque[Front] & Deque[Front + 1]) - halfplanes[i].S) ^ (halfplanes[i].T - halfplanes[i].S)) > eps) {
            Front++;
        }
        Deque[Tail] = halfplanes[i];
        Tail++;
    }
}
```

```

        Front++;
    }
    Deque[++Tail] = halfplanes[i];
}
while (Front < Tail && (((Deque[Tail] & Deque[Tail -
↪ 1]) - Deque[Front].S) ^ (Deque[Front].T -
↪ Deque[Front].S)) > eps) {
    Tail--;
}
while (Front < Tail && (((Deque[Front] & Deque[Front -
↪ 1]) - Deque[Tail].S) ^ (Deque[Tail].T -
↪ Deque[Tail].T)) > eps) {
    Front++;
}
if (Tail <= Front + 1) {
    return false;
}
return true;
}

// 获取半平面交内核点集 Res
void GetHalfPlaneInsertConvex() {
    int Cnt = 0;
    for (int i = Front; i < Tail; ++i) {
        Res[Cnt++] = Deque[i] & Deque[i + 1];
    }
    if (Front < Tail - 1) {
        Res[Cnt++] = Deque[Front] & Deque[Tail];
    }
}
};

```

## 6.2 Stereoscopic

```

#include <bits/stdc++.h>

const double INF = 1e20;
const int maxn = "Edit";
const double eps = 1e-8;
const double delta = 0.98;

// 点
struct Point {
    double X, Y, Z;

```

```

    void Input() {
        scanf("%lf%lf%lf", &X, &Y, &Z);
    }
};

// 求点 A、B 间距离
double Distance(Point A, Point B) {
    return sqrt((A.X - B.X) * (A.X - B.X) + (A.Y - B.Y) * (A.Y
        ↪ - B.Y) + (A.Z - B.Z) * (A.Z - B.Z));
}

int N;
Point points[maxn];

// 模拟退火求 N 个点的最小球覆盖
double MinimimSphereCoverage() {
    Point Cur = points[0];
    double Probability = 10000, Ans = INF;
    while (Probability > eps) {
        int Book = 0;
        for (int i = 0; i < N; ++i) {
            if (Distance(Cur, points[i]) > Distance(Cur,
                ↪ points[Book])) {
                Book = i;
            }
        }
        double RADIUS = Distance(Cur, points[Book]);
        Ans = min(Ans, RADIUS);
        Cur.X += (points[Book].X - Cur.X) / RADIUS *
            ↪ Probability;
        Cur.Y += (points[Book].Y - Cur.Y) / RADIUS *
            ↪ Probability;
        Cur.Z += (points[Book].Z - Cur.Z) / RADIUS *
            ↪ Probability;
        Probability *= delta;
    }
    // 返回覆盖最小球半径
    return Ans;
}

```

## 7 Others

### 7.1 Factorial

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

void Factorial() {
    int res[10010];
    int Book = 1;
    int BaoFour = 0;
    res[Book] = 1;
    int n;
    scanf("%d", &n);
    // 乘法计算
    for (int i = 1; i <= n; ++i) {
        BaoFour = 0;
        for (int j = 1; j <= Book; ++j) {
            res[j] = res[j] * i + BaoFour;
            BaoFour = res[j] / 10000;
            res[j] = res[j] % 10000;
        }
        if (BaoFour > 0) {
            res[++Book] += BaoFour;
        }
    }
    printf("%d", res[Book]);
    // 补零输出
    for (int i = Book - 1; i > 0; --i) {
        if (res[i] >= 1000) {
            printf("%d", res[i]);
        }
        else if (res[i] >= 100) {
            printf("0%d", res[i]);
        }
        else if (res[i] >= 10) {
            printf("00%d", res[i]);
        }
        else {
            printf("000%d", res[i]);
        }
    }
    putchar('\n');
}
```

## 7.2 FastIO

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

```
// 普通读入挂
```

```
template <class T>
inline bool read(T &ret) {
    char c;
    int sgn;
    if (c = getchar(), c == EOF) {
        return false;
    }
    while (c != '-' && (c < '0' || c > '9')) {
        c = getchar();
    }
    sgn = (c == '-') ? -1 : 1;
    ret = (c == '-') ? 0 : (c - '0');
    while (c = getchar(), c >= '0' && c <= '9') {
        ret = ret * 10 + (c - '0');
    }
    ret *= sgn;
    return true;
}
```

```
// 普通输出挂
```

```
template <class T>
inline void out(T x) {
    if (x < 0) {
        putchar('-');
        x = -x;
    }
    if (x > 9) {
        out(x / 10);
    }
    putchar(x % 10 + '0');
}
```

```
// 牛逼读入挂
```

```
namespace FastIO {
    const int MX = 4e7;
    char buf[MX];
    int c, sz;
    void begin() {
        c = 0;
    }
}
```

```
        sz = fread(buf, 1, MX, stdin);
    }
    template <class T>
    inline bool read(T &t) {
        while (c < sz && buf[c] != '-' && (buf[c] < '0' ||
        ↪ buf[c] > '9')) {
            c++;
        }
        if (c >= sz) {
            return false;
        }
        bool flag = 0;
        if (buf[c] == '-') {
            flag = 1;
            c++;
        }
        for (t = 0; c < sz && '0' <= buf[c] && buf[c] <= '9';
        ↪ ++c) {
            t = t * 10 + buf[c] - '0';
        }
        if (flag) {
            t = -t;
        }
        return true;
    }
};

// 超级读写挂
namespace IO{
    #define BUF_SIZE 100000
    #define OUT_SIZE 100000
    #define ll long long
    //fread->read

    bool IOerror=0;
    inline char nc(){
        static char
        ↪ buf[BUF_SIZE], *p1=buf+BUF_SIZE, *pend=buf+BUF_SIZE;
        if (p1==pend){
            p1=buf; pend=buf+fread(buf,1,BUF_SIZE,stdin);
            if (pend==p1){IOerror=1;return -1;}
            //{printf("IO error!\n");system("pause");for
            ↪ (;;);exit(0);}
        }
    }
}
```



```

    }
    return *p1++;
}
inline bool blank(char ch){return ch=='
↪ ' || ch=='\n' || ch=='\r' || ch=='\t';}
inline void read(int &x){
    bool sign=0; char ch=nc(); x=0;
    for (;blank(ch);ch=nc());
    if (IError)return;
    if (ch=='-')sign=1,ch=nc();
    for (;ch>='0'&&ch<='9';ch=nc())x=x*10+ch-'0';
    if (sign)x=-x;
}
inline void read(ll &x){
    bool sign=0; char ch=nc(); x=0;
    for (;blank(ch);ch=nc());
    if (IError)return;
    if (ch=='-')sign=1,ch=nc();
    for (;ch>='0'&&ch<='9';ch=nc())x=x*10+ch-'0';
    if (sign)x=-x;
}
inline void read(double &x){
    bool sign=0; char ch=nc(); x=0;
    for (;blank(ch);ch=nc());
    if (IError)return;
    if (ch=='-')sign=1,ch=nc();
    for (;ch>='0'&&ch<='9';ch=nc())x=x*10+ch-'0';
    if (ch=='.'){
        double tmp=1; ch=nc();
        for
            ↪ (;ch>='0'&&ch<='9';ch=nc())tmp/=10.0,x+=tmp*(ch-'0');
    }
    if (sign)x=-x;
}
inline void read(char *s){
    char ch=nc();
    for (;blank(ch);ch=nc());
    if (IError)return;
    for (;!blank(ch)&&!IError;ch=nc())*s++=ch;
    *s=0;
}
inline void read(char &c){
    for (c=nc();blank(c);c=nc());
    if (IError){c=-1;return;}
}

```



```

11 x1=(11)floor(x); if (x-floor(x)>=0.5)++x1;
11 x2=x1/mul[y],x3=x1-x2*mul[y]; print(x2);
if (y>0){out('.'); for (size_t
    ↪ i=1;i<y&& x3*mul[i]<mul[y];out('0'),++i);
    ↪ print(x3);}
}
void println(double x,int y){print(x,y);out('\n');}
void print(char *s){while (*s)out(*s++);}
void println(char *s){while (*s)out(*s++);out('\n');}
void flush(){if
    ↪ (p1!=buf){fwrite(buf,1,p1-buf,stdout);p1=buf;}}
~Ostream_fwrite(){flush();}
}Ostream;
inline void print(int x){Ostream.print(x);}
inline void println(int x){Ostream.println(x);}
inline void print(char x){Ostream.out(x);}
inline void println(char
    ↪ x){Ostream.out(x);Ostream.out('\n');}
inline void print(ll x){Ostream.print(x);}
inline void println(ll x){Ostream.println(x);}
inline void print(double x,int y){Ostream.print(x,y);}
inline void println(double x,int y){Ostream.println(x,y);}
inline void print(char *s){Ostream.print(s);}
inline void println(char *s){Ostream.println(s);}
inline void println(){Ostream.out('\n');}
inline void flush(){Ostream.flush();}
#undef ll
#undef OUT_SIZE
#undef BUF_SIZE
};
using namespace IO;

```

### 7.3 LeapYear

```

#include <bits/stdc++.h>

inline bool Leap(int Year) {
    return (!(Year % 4) && (Year % 100)) || !(Year % 400);
}

```

### 7.4 NimGame

```

#include <bits/stdc++.h>

```

```
// 尼姆博弈
bool Nim(std::vector<int> Num) {
    int Ans = 0;
    for (int i = 0; i < int(Num.size()); ++i) {
        Ans ^= Num[i];
    }
    // ans 不为零则先手赢，否则为后手赢
    return Ans != 0 ? true : false;
}
```

## 7.5 vim

```
syntax on
set nu
set tabstop=4
set shiftwidth=4
set cindent
set mouse=a

map <F9> :call Run(<CR>

func! Run()
    exec "w"
    if &filetype == "cpp"
        exec "!g++ -Wall % -o %<"
        exec "! ./%<"
    elseif &filetype == "python"
        exec "!python3 %"
    endif
endfunc
```