ICPC Team Notebook

Team: HCMUT Apollo

TABLE OF CONTENT:

I. CẤU TRÚC DỮ LIỆU CƠ BẢN

1. Deque và bài toán Min Range trong đoạn tịnh tiến

- **Deque tương tự queue nhưng nó có thể thêm và xóa ở cả hai đầu.**
  (push_front(x) ; push_back(x); pop_front(); pop_back())

Problem: Cho dãy A có N phần tử và số nguyên dương k, với mỗi $i \geq k$, tìm giá trị nhỏ nhất của các phần tử thuộc mảng A trong đoạn $[i-k+1, i]$.

Hướng giải:

- Duy trì một deque chứa index các phần tử của A trong đó các phần từ luôn tăng từ đầu đến cuối deque.
- Phần tử đầu tiên của deque luôn có chỉ số lớn hơn hoặc bằng $i-k+1$.
- Khi thêm $A[i]$ vào cuối deque, nếu phần từ cuối hiện tại lớn hơn $A[i]$ thì pop nó ra và tiếp tục kiểm tra nhằm duy trì một deque tăng. MinRange[i] và giá trị đầu của deque.

Code:

```
deque <int> dq;

/* Làm rỗng deque */
while (dq.size()) dq.pop_front();

/* Duyệt lần lượt các phần tử từ 1 đến N */
for (int i = 1; i <= N; ++i) {
    /* Loại bỏ các phần tử có giá trị lớn hơn hoặc bằng A[i] */
    while (dq.size() && A[dq.back()] >= A[i]) dq.pop_back();

    /* Đẩy phần tử i vào queue */
    dq.push_back(i);

    /* Nếu phần tử đầu tiên trong deque nằm ngoài khoảng tính
       thì ta sẽ loại bỏ ra khỏi deque */
    if (dq.front() + k <= i) dq.pop_front();

    /* minRange[i] là giá trị nhỏ nhất trong đoạn [i - k + 1 … i] */
    if (i >= k) minRange[i] = A[dq.front()];
}
```

2. Stack và bài toán tìm phần tử lớn hơn gần nhất

- **Các method của stack STL:** (empty(); size(); top(); push(x); pop();)

Problem: Cho một dãy H có n phần tử thể hiện chiều cao của n người trong hàng ngang tương ứng. Với mỗi $1 \leq i \leq n$, tìm người gần i nhất về bên trái mà người đó cao hơn hoặc bằng i .

Hướng giải:

- Ta duyệt lần lượt từ trái sang và duy trì một stack chứa các phần tử trong H theo index sao cho giá trị của chúng giảm dần.
- Khi gặp một phần tử mới thứ i, ta sẽ pop() đến khi nào top() là phần tử mang giá trị lớn hơn H[i], rõ ràng đây là giá trị cần tìm ứng với i. Nếu stack bị pop() hết thì ko có ai cao hơn i về phía bên trái. Cuối cùng thêm i vô stack.

Code:

```
stack<int> st;

for (int i = 1; i <= n; ++i) {
    while (!st.empty() && a[st.top()] <= a[i]) st.pop();
    int ans = -1;
    if (!st.empty())
        ans = st.top();
    cout << ans << ' ';
    st.push(i);
}
```

3. Set và multiset

Method của set: (insert(x); begin(); end(); erase(<iterator>); erase(x); clear(); find(x); upper_bound(x), lower_bound(x), euqal_range(x) …)

Custom set's comparator:

```
auto keycmp= [] (const T& a, const T&b) -> bool { /* logic code */ };
set<T,decltype(keycmp)> st(keycmp);
```

Note: Cẩn thận tránh iterator invalidation (xóa phần tử iterator trỏ tới nhưng sau đó vẫn thực hiện tính toán trên iterator đó). Method equal range return pair of interator (boundary of all elements have the same value x)

4. Map và multimap

Method của map: (insert(x); count(); equal_range(x); erase(x); begin(), end(), find(x); upper_bound(), lower_bound(), …)

Custom map's comparator:

```
auto keycmp=[] (const int& a, const int& b) -> bool { return a>b; };
map<int,int,decltype(keycmp)> mp(keycmp);
```

5. Custom priority_queue

Code:

```
auto kcmp=[&] (int a, int b) -> bool { return t[a] > t[b]; };
priority_queue<int, vector<int>, decltype(kcmp)> q(kcmp);
```

Note: Với hàm lambda compare dạng a.data > b.data tức là queue sẽ ưu tiên từ bé đến lớn chứ không phải từ lớn đến bé.

6. String

Some method:

```
string::substr:  str.substr ( size_t pos, size_t len ).
string::find : str.find ( const string& s, size_t pos ). Return string::npos if s is not exist.
string::replace : str.replace ( size_ t pos, size_t len, const string& s ).
string::find_first_of: str.find_first_of ( const strign &s, size_t pos ). Searches the string for  the first character that matches any of the characters specified in its arguments.
string::erase: str.erase ( size_t pos, size_t len )
```

7. Disjoint set union

- Cấu trúc dữ liệu lưu thông tin của các phần tử theo nhóm. Với hai thao tác chủ yếu là union_set(int a, int b) và find_parent(int a).
- Ta dùng mảng parent[] và sz[] cho DSU cơ bản và lưu thêm thông tin từng thành phần nếu cần. Parent[i] là node cha của i, sz[i] với i là node gốc bằng kích thước của thành phần i hiện tại. Các thao tác khi tối ưu có độ phức tạp gần như O(1).

Cài đặt:

```
void make_set(int v) {
    parent[v] = v;
    sz[v] = 1; // Ban đầu tập hợp chứa v có kích cỡ là 1
}
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (sz[a] < sz[b]) swap(a, b); // Đặt biến a là gốc của cây có kích cỡ lớn hơn
        parent[b] = a;
        sz[a] += sz[b]; // Cập nhật kích cỡ của cây mới gộp lại
    }
}

int find_set(int v) {
    return v == parent[v] ? v : parent[v] = find_set(parent[v]);
}
```

8. Segment tree

Segment tree là một cây quản lý thông tin theo đoạn, mỗi node sẽ quản lý một đoạn [l,r]. Node 1 là gốc sẽ quản lý đoạn [1,N]. Với st[node] quản lý đoạn [l,r], st[node*2] quản lý đoạn [l,(l+r)/2] và st[node*2+1] quản lý đoạn [(l+r)/2+1,r] .

Cài đặt với bài toán Min Range:

```
int n, q;
int a[maxN];
int st[4 * maxN];

void build(int id, int l, int r) {
    if (l == r) {
        st[id] = a[l];
        return;
    }
    int mid = l + r >> 1; // (l + r) / 2
    build(2 * id, l, mid);
    build(2 * id + 1, mid + 1, r);

    st[id] = min(st[2 * id], st[2 * id + 1]);
}

void update(int id, int l, int r, int i, int val) {
    if (l > i || r < i) return;

    if (l == r) {
        st[id] = val;
        return;
    }

    int mid = l + r >> 1; // (l + r) / 2
    update(2 * id, l, mid, i, val);
    update(2 * id + 1, mid + 1, r, i, val);

    st[id] = min(st[2 * id], st[2 * id + 1]);
}


int get(int id, int l, int r, int u, int v) {
    if (l >  v || r <  u) return inf;

    if (l >= u && r <= v) return st[id];

    int mid = l + r >> 1; // (l + r) / 2
    int get1 = get(2 * id, l, mid, u, v);
    int get2 = get(2 * id + 1, mid + 1, r, u, v);

    return min(get1, get2);
}
```

Lazy propagation cho segment tree với bài toán Sum Range

Với lazy propagation, khi update một đoạn [ss,ed], ta sẽ trì hoãn việc update các node quản lý đoạn [l,r] mà [l,r] nằm trong [ss,ed] (vì thông thường các node này sẽ được update đồng đều nguyên cả đoạn với nhau). Sau đó ta lưu thông tin cần update của st[node] vào lz[node]. Trong tương lai, khi update hay getSum, nếu đi qua đoạn có lz[node] cần update, ta update st[node], truyền thông tin lz[node] cho lz[node*2] và lz[node*2+1], set lz[node] về giá trị rỗng.

```
void updateRange( ll l, ll r, ll node, ll ss, ll ed, ll inc  ) {
    if(  ed<l || r<ss || l>r ) return;

    if( lz[node]!=0 ) {
        if(l!=r) {
            lz[2*node]+=lz[node];
            lz[2*node+1]+=lz[node];
        }
        st[node]+=lz[node]*(ll)(r-l+1);
        lz[node]=0;
    }
```

```
    if( ss<=l && r<=ed ) {
        lz[node]+=inc;
        return;
    }

    if( ss<=l ) st[node]+=(ed-l+1)*inc;
    else if( r<=ed ) st[node]+=(r-ss+1)*inc;
    else st[node]+=(ed-ss+1)*inc;

    updateRange(l, (r+l)/2, 2*node, ss, ed, inc );
    updateRange( (r+l)/2+1, r, 2*node+1, ss, ed, inc );

}
```

(với getsum thì cũng xử lý lz[node] trước khi lấy sum)

9. SQRT decomposition và Mo Algorithm

Chia mảng cần trả lời truy vấn thành $\sqrt{n}$ đoạn con, mỗi đoạn gồm khoảng $\sqrt{n}$ phần tử. Khi trả lời querry dạng câu hỏi trên đoạn [l,r], chúng ta duyệt và lấy thông tin qua tất cả đoạn con nằm trong [l,r], và đồng thời lấy thêm các phần tử nằm hai rìa bị thừa ra. Độ phức tạp cho mỗi truy vấn sẽ là $O\left(\sqrt{n}\right)$. Truy vấn update trên từng phần tử sẽ tốn O(1).

Problem: Cho mảng A gồm N phần tử. Trả lời Q truy vấn dạng (l,r,k) yêu cầu đếm số phần tử của A trong đoạn [l,r] có giá trị bằng k.

Code:

```
const int BLOCK_SIZE = 320;
const int N = 1e5 + 2;

int n;
int cnt[N / BLOCK_SIZE + 2][N];
int a[N];

void preprocess()
{
    for (int i = 0; i < n; ++i)
        ++cnt[i / BLOCK_SIZE][a[i]];
}
int query(int l, int r, int k)
{
    int blockL = (l + BLOCK_SIZE - 1) / BLOCK_SIZE;
    int blockR = r / BLOCK_SIZE;
    if (blockL >= blockR)
        return count(a + l, a + r + 1, k);

    int sum = 0;
    for (int i = blockL; i < blockR; ++i)
        sum += cnt[i][k];

    for (int i = l, lim = blockL * BLOCK_SIZE; i < lim; ++i)
        if (a[i] == k) ++sum;

    for (int i = blockR * BLOCK_SIZE; i <= r; ++i)
        if (a[i] == k) ++sum;

    return sum;
}
void update(int u, int v)
{
    int block = u / BLOCK_SIZE;
    --cnt[block][a[u]];
    a[u] = v;
    ++cnt[block][a[u]];
}
```

Mo algorithm:

Đối với các bài toán trả lời truy vấn trên đoạn [l,r], nếu từ [l,r] ta xác định được đáp án querry trên $[l \pm 1, r]$ hoặc $[l, r \pm 1]$, khi đó ta có thể áp dụng Mo algorithm để sort các truy vấn và đạt được thời gian $O(N\sqrt{Q} + Q\sqrt{N})$.

Problem: Cho dãy A gồm N phần tử. Thực hiện Q truy vấn, mỗi truy vấn [l,r] yêu cầu tìm $mode(A_l, A_{l+1}, ...A_r)$ = giá trị xuất hiện nhiều lần nhất. $N, Q, A_i \le 10^5$.

Hướng giải: Sau khi trả lời truy vấn $[l_1, r_1]$, để trả lời truy vấn $[l_2, r_2]$, ta chỉ cần lần lượt thay đổi mảng đếm cnt. Nếu $l_2 > l_1$ thì giảm số lần xuất hiện của $A_{l_1}, A_{l_1+1}, ...A_{l_2}$, nếu $l_2 < l_1$ thì tăng. Đối với r cũng tương tự.

Thực hiện xếp lại truy vấn, ta chia đoạn $[1, N]$ thành các block có size = $\sqrt{N}$. Khi so sánh hai truy vấn, nếu đầu trái nằm ở hai block khác nhau thì xếp theo giá trị đầu trái. Nếu cùng block thì xếp theo giá trị r tăng dần.

Code:

```
int s=(int)sqrt(n);
sort(qry.begin(),qry.end(), [&] (qr &a, qr& b) {
     if(a.l/s!=b.l/s) return a.l<b.l;
     else if((a.l/s)%2==0) return a.r<b.r;
     else return a.r>b.r;}  );
```

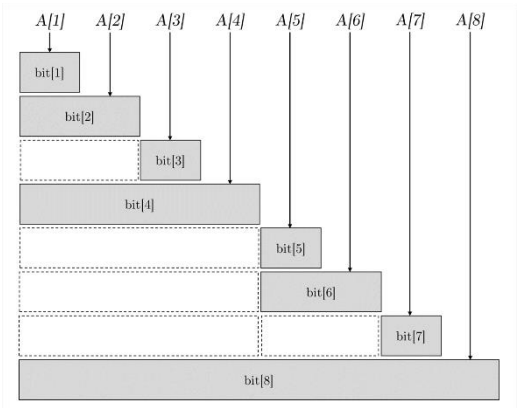
10. Cây chỉ số nhị phân (Fenwick Tree – Binary Index Tree)

Problem: Bài toán Min Range Querry (update giá trị phần tử hoặc lấy tổng đoạn [1,r]).

Hướng giải sử dụng BITree: Mỗi số có thể biểu diễn dưới dạng nhị phân hay là tổng của các 2^i. Khi đó, ta sẽ dùng mỗi phần tử của BIT, chẳng hạn BIT[m] để lưu tổng của 2^k phần tử (k là bit set đầu tiên của m tính từ 0). Khi đó để tính tổng đoạn $[1, r]$, ta chỉ cần tính tổng các đoạn $[2^{i_1}+1; 2^{i_1}+2^{i_2}]$.

Cài đặt:

```
int getSum(int p) {
    int idx = p, ans = 0;
    while (idx > 0) {
        ans += bit[idx];
        idx -= (idx & (-idx));
    }
    return ans;
}
void update(int u, int val) {
    int idx = u;
    while (idx <= n) {
        bit[idx] += val;
        idx += (idx & (-idx));
    }
}
```



Getsum, ví dụ là với đoạn [1,11], 11=1011, sẽ bằng bit[1011]+bit[1010]+bit[1000]. Bởi bit[1011] chỉ quản lí 1 phần tử là chính nó. bit[1010] quản lí 2 phần tử và bit[1000] quản lý 8 phần tử đầu tiên.

Update hơi khó hiểu hơn. Ví dụ update 21=10101, khi đó ta sẽ cộng thêm vào các bit[10101], bit[10110], bit[11000]. Việc chứng minh xem như bài tập cho bạn đọc =)).

Problem: Bài toán Sum range querry với việc update tăng giá trị phần tử cho cả một đoạn. Trả lời truy vấn trên đoạn.

Ta duy trì hai BIT B1 và B2. Khi updateRange(u,v,val) thì ta sẽ dùng hàm update trên như sau: update(B1, u, x), update(B1, v+1, - x), update(B2, u , x*(u-1)), update(B2, v+1, - x*v).

Ta thấy rằng thực chất mảng B1 sẽ trả lời truy vấn trên điểm (giá trị một điểm) khi gọi getSum(B1,k). Vì từ v+1 trở đi x sẽ triệt tiêu với -x. Kết hợp với mảng B2, ta sẽ tính được sum thực sự như sau:

$$prefixSum[i] = getSum[B1, i] * i - getSum[B2, i]$$

11. Sparse table

Áp dụng trong bài toán Min range querry hoặc một số bài toán mà việc lấy thông tin đoạn [l,r] có thể truy xuất từ các 2 đoạn con (có thể cắt nhau).

Ta sử dụng mảng $M[0, N-1][0, \log N]$ với $M[i][j]$ lưu thông tin cần trả lời của đoạn $[i, i+2^j-1]$. Việc xây dựng $M[i][j]$ sẽ phục thuộc vào hai đoạn con $M[i][j-1]$ và $M[i+2^{j-1}][j-1]$.

Code với bài Min range:

```
for (i = 0; i < N; i++)
    M[i][0] = i;
  // Tính M với các khoảng dài 2^j
for (j = 1; 1 << j <= N; j++)
    for (i = 0; i + (1 << j) - 1 < N; i++)
      if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
        M[i][j] = M[i][j - 1];
      else
        M[i][j] = M[i + (1 << (j - 1))][j - 1];
```

12. Bitset

A bitset is an array of bools but each boolean value is not stored in a separate byte instead, bitset optimizes the space such that each boolean value takes 1-bit space only

Khai báo: bitset<size> varname;

Member function: set(id,val) _ set value ở id, vd bs.set(1,0); flip() _ đảo ngược bit ở index; count() _ đếm số set bit, any() _ kiểm tra nếu tồn tại bit set; none() _ kiểm tra nếu ko tồn tại bit set; all() _ kiểm tra nếu tất cả là bit set; to_ulong() _ chuyển bitset thành unsigned long. Và bset._Find_next(index) trả về vị trí bit set tiếp theo kể từ vị trí index, bset._Find_firt() trả về vị trí đầu tiên.

Operation: [], &, |, ^, ~ ...

13. Matrix:

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

using namespace std;

using type = int; // Kiểu dữ liệu các phần tử của ma trận

struct Matrix {
    vector <vector <type> > data;

    // Số lượng hàng của ma trận
    int row() const { return data.size(); }

    // Số lượng hàng của ma trận
    int col() const { return data[0].size(); }

    auto & operator [] (int i) { return data[i]; }

    const auto & operator[] (int i) const { return data[i]; }

    Matrix() = default;

    Matrix(int r, int c): data(r, vector <type> (c)) { }

    Matrix(const vector <vector <type> > &d): data(d) {

        // Kiểm tra các hàng có cùng size không và size có lớn hơn 0 hay không
        // Tuy nhiên không thực sự cần thiết, ta có thể bỏ các dòng /**/ đi
        /**/ assert(d.size());
        /**/ int size = d[0].size();
        /**/ assert(size);
        /**/ for (auto x : d) assert(x.size() == size);
    }
```

```
    // In ra ma trận.
    friend ostream & operator << (ostream &out, const Matrix &d) {
        for (auto x : d.data) {
            for (auto y : x) out << y << ' ';
            out << '\n';
        }
        return out;
    }

    // Ma trận đơn  vị
    static Matrix identity(long long n) {
        Matrix a = Matrix(n, n);
        while (n--) a[n][n] = 1;
        return a;
    }

    // Nhân ma trận
    Matrix operator * (const Matrix &b) {
        Matrix a = *this;

        // Kiểm tra điều kiện nhân ma trận
        assert(a.col() == b.row());

        Matrix c(a.row(), b.col());
        for (int i = 0; i < a.row(); ++i)
            for (int j = 0; j < b.col(); ++j)
                for (int k = 0; k < a.col(); ++k)
                    c[i][j] += a[i][k] * b[k][j];
        return c;
    }
    // Lũy thừa ma trận
    Matrix pow(long long exp) {

        // Kiểm tra điều kiện lũy thừa ma trận (là ma trận vuông)
        assert(row() == col());

        Matrix base = *this, ans = identity(row());
        for (; exp > 0; exp >>= 1, base = base * base)
            if (exp & 1) ans = ans * base;
        return ans;
    }
};
```

14. Big Interger

```
class bigInt {
public:
    int leng=0;
    string dig="";
    bigInt(string s) {
        reverse(s.begin(),s.end());
        dig=s;
        leng=s.length();
    }
    bigInt(long long a) {
        string s=to_string(a);
        reverse(s.begin(),s.end());
        dig=s;
        leng=s.length();
    }
    bigInt() {};
};

bigInt operator+( const bigInt &a, const bigInt&b ) {
    bigInt temp;
    int la=a.leng, lb=b.leng;
    int note=0;
    for(int i=0; i<max(la,lb); i++) {
        int v=0;
        if(i<la) v+=a.dig[i]-48;
        if(i<lb) v+=b.dig[i]-48;
        temp.dig+=(char)((v+note)%10+48);
        note=(v+note)/10;
    }
```

```
    if(note>0) temp.dig+=(char)(note+48);
    temp.leng=temp.dig.length();
    return temp;
}

bigInt operator*(const bigInt &a, const bigInt &b) {
    int n=a.leng, m=b.leng;
    vector<int> aux(n+m,0);
    for(int i=0; i<n; i++) {
        for(int j=0; j<m; j++) {
            aux[i+j]+=((int)a.dig[i]-'0')*((int)b.dig[j]-'0');
        }
    }

    bigInt ans;
    for(int i=0; i<n+m-1; i++) {
        ans.dig+=char( aux[i]%10+(int)'0' );
        aux[i+1]+=aux[i]/10;
    }
    if(aux[n+m-1]!=0) ans.dig+=char( aux[n+m-1]%10+(int)'0' );
    return ans;
}

ostream& operator<<(ostream& os, const bigInt &a) {
    string temp=a.dig;
    reverse(temp.begin(),temp.end());
    os << temp;
    return os;
}
```

15. Policy based data structures

Library and namespace.

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace __gnu_cxx;
typedef tree<ll,null_type,less<ll>,rb_tree_tag,tree_order_statistics_node_update> ordered_set;
```

Ordered_set này hỗ trợ các method tương tự như set, và hỗ trợ thêm 2 method quan trọng: find_by_order() và order_of_key(). Cái thứ nhất trả vè iterator tới số lớn thứ k (đếm từ 0). Cái thứ 2 trả về số phần tử trong set nhỏ hơn key.

```
ordered_set X;
    X.insert(1);
    X.insert(2);
    X.insert(4);
    X.insert(8);
    X.insert(16);

    cout<<*X.find_by_order(1)<<endl; // 2
    cout<<*X.find_by_order(2)<<endl; // 4
    cout<<*X.find_by_order(4)<<endl; // 16
    cout<<(end(X)==X.find_by_order(6))<<endl; // true

    cout<<X.order_of_key(-5)<<endl;  // 0
    cout<<X.order_of_key(1)<<endl;   // 0
    cout<<X.order_of_key(3)<<endl;   // 2
    cout<<X.order_of_key(4)<<endl;   // 2
    cout<<X.order_of_key(400)<<endl; // 5
```

Lưu ý: policy based structure chỉ được support bởi g++ compiler, không phải là một phần của c++ standard.

II. CÁC CHỦ ĐỀ TRÊN ĐỒ THI

1. Bài toán Lowest Common Ancestor – LCA

Problem: Cho cây có gốc là 1. Với mỗi truy vấn dạng a, b, tìm tổ tiên chung gần nhất của hai node đó.

Hướng giải sử dụng Binary Lifting (sparse table): Khởi tạo mảng par[0,logN][1,N] với par[i][j] là tổ tiên thứ 2^i của node j. Khi đó par[i][j]=par[i-1][par[i-1][j]]. Từ đó có thể tình tổ tiên thứ k của node j trong thời gian O(log). Kết hợp binary search để tìm được LCA.

Code:

Hàm tìm tổ tiên thứ k

```
// Find k-th ancestor of node
int ancestor( vector<vector<int>> &par, int node, int k) {
    int i=0;
    while(k>0) {
        if(k%2==1) node=par[i][node];
        k=k>>1;
        i++;
    }
    return node;
}
```

Trong hàm chính:

```
// DFS to initialize all value of array h and par[0];
process(edg, par, h, 1);
// Now, par[i][j]=par[i-1][par[i-1][j]] if( h[j]>=2^j )
int p2=1;
for__(1,21) {
    p2*=2;
    for(int j=1; j<=n; j++) {
        if(h[j]>=p2) par[i][j]=par[i-1][par[i-1][j]];
        else par[i][j]=-1;
    }
}
cin >> q;
while(q--) {
    int a, b;
    cin >> a >> b;
    if(a==b) cout << a << el;
    else {
        if (h[a]<h[b]) swap(a,b);
        int lo=0, hi=h[a];
        int mid=(lo+hi+1)/2;
        while(lo<hi) {
            int aa=ancestor(par,a,h[a]-mid), ab=ancestor(par,b,h[b]-mid);
            if(aa==ab) lo=mid;
            else hi=mid-1;
            mid=(lo+hi+1)/2;
        }
        cout << ancestor(par,a,h[a]-lo) << el;
    }
}
```

2. Breadth First Search

Code:

```
    queue <int> q; q.push(s);
    visit[s] = true;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : g[u]) {
            if (!visit[v]) {
                d[v] = d[u] + 1; par[v] = u;
                visit[v] = true;
                q.push(v);
            }
        }
    }
```

3. Cây DFS và bài toán đỉnh khớp, cạnh cầu

Tính chất: Trong đồ thị vô hướng, không tồn tại cung chéo (cung có dạng u->v trong đó u, v thuộc hai nhánh khác nhau của cây DFS). Cạnh thuộc cây DFS gọi là span-edge. Cạnh không thuộc cây DFS gọi là back-edge.

Problem: Cho đồ thị G liên thông, tìm các cạnh cầu (cạnh mà khi bỏ đi thì số thành phần liên thông tăng lên) và các đỉnh khớp (đỉnh mà khi bỏ đi thì thì số thành phần liên thông tăng lên).

Hướng giải: Đầu tiên, một span-edge uv là cạnh cầu khi và chỉ khi không tồn tại back-edge nào nối giữa con cháu của cạnh uv và tổ tiên của cạnh uv. Thứ hai, một back-edge không bao giờ là một cạnh cầu.

Vậy với mỗi span-edge trên đồ thị, ta sẽ tìm xem có back-edge nào vượt qua nó không. Ta định nghĩa mảng num[] cho biết thứ tự duyệt tới đỉnh u trong dfs tree. Và mảng low[] được định nghĩa như sau:

$$low[u] = min\begin{cases} num[u] \\ num[p] \;\; với\ mọi\ p\ sao\ cho\ (u,p)\ là\ back\ edge \\ low[v]\ với\ mọi\ v\ sao\ cho\ (u,v)\ là\ span\ edge \end{cases}$$

Span-edge (u,v) là cạnh cầu khi và chỉ khi (low[v] > num[u] hoặc low[v]==num[v]). Nếu low[v] < num[v] thì tồn tại một back-edge nối từ một đỉnh trong cây con gốc 'v' tới một đỉnh thuộc tổ tiên của 'v'.

Xét đỉnh u, và v là con của u trong dfs tree. Nếu với mọi v, low[v]<num[v] (có một back-edge nối từ dưới lên) thì u không thể là đỉnh khớp. Ngược lại, nếu u không phải là gốc và tồn tại đỉnh con v sao cho (low[v]==num[v] hay low[v] ≥ num[u]) , trong trường hợp u là gốc, dễ thấy nếu u có 2 con trở lên thì u là đỉnh khớp.

Code:

```
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> num, low;
int timer = 1;

void dfs(int u, int p = -1) { // v is node, p is pre_node, or parent
    visited[u] = true;
    num[u] = low[u] = timer++;
    int child=0;
    for (int v : adj[u]) {
        if (v == p) continue;
        if (visited[v]) low[u] = min(low[u], num[v]);
        else {
            dfs(v, u);
             child++;
            low[u] = min(low[u], low[v]);
            if (low[v] > num[u]){
                // Now (u,v) is brigde, do something here …
             }
             if(u == p && child>1 ) {} // u is root and is cut vertice, do something …
             else if(low[v] >= num[u]) {} // u is cut vertice, do something …
        }
    }
}
```

4. Thuật toán Kruskal và bài toán tìm cây khung nhỏ nhất

Một vài tính chất của cây khung nhỏ nhất trong đồ thị vô hướng có trọng số:

1. *Tính chất chu trình:* Trong một chu trình C bất kỳ, nếu e là cạnh có trọng số lớn nhất tuyệt đối (không có cạnh nào có trọng số bằng e) thì e không thể nằm trên bất kỳ cây khung nhỏ nhất nào.

2. *Đường đi hẹp nhất:* Xét 2 đỉnh u, v bất kỳ trong đồ thị. Nếu w là trọng số của cạnh lớn nhất trên đường đi từ u đến v trên cây khung nhỏ nhất của đồ thị thì ta không thể tìm được đường đi nào từ u đến v trên đồ thị ban đầu chỉ đi qua những cạnh có trọng số nhỏ hơn w.
3. *Tính duy nhất:* Nếu tất cả các cạnh đều có trọng số khác nhau thì chỉ có duy một cây khung nhỏ nhất. Ngược lại, nếu một vài cạnh có trọng số giống nhau thì có thể có nhiều hơn một cây khung nhỏ nhất.

4. *Tính chất cạnh nhỏ nhất:* Nếu e là cạnh có trọng số nhỏ nhất của đồ thị, và không có cạnh nào có trọng số bằng e thì e nằm trong mọi cây khung nhỏ nhất của đồ thị.

Thuật toán kruskal: Ban đầu mỗi đỉnh là một cây riêng biệt, ta tìm cây khung nhỏ nhất bằng cách duyệt các cạnh theo trọng số từ nhỏ đến lớn, rồi hợp nhất các cây lại với nhau.

Giả sử ta cần tìm cây khung nhỏ nhất của đồ thị G. Thuật toán bao gồm các bước sau:

- Khởi tạo rừng F (tập hợp các cây), trong đó mỗi đỉnh của G tạo thành một cây riêng biệt.
- Khởi tạo tập S chứa tất cả các cạnh của G.
- Chừng nào S còn khác rỗng và F gồm hơn một cây
  - Xóa cạnh nhỏ nhất trong S
  - Nếu cạnh đó nối hai cây khác nhau trong F, thì thêm nó vào F và hợp hai cây kề với nó làm một
  - Nếu không thì loại bỏ cạnh đó.

Khi thuật toán kết thúc, rừng chỉ gồm đúng một cây và đó là một cây khung nhỏ nhất của đồ thị G

Code:

```
#include <bits/stdc++.h>
using namespace std;

struct Edge {
    int u, v, c; // u, v là 2 đỉnh, c là trọng số cạnh
    Edge(int _u, int _v, int _c): u(_u), v(_v), c(_c) {};
};

struct Dsu {
    vector<int> par;

    void init(int n) {
        par.resize(n + 5, 0);
        for (int i = 1; i <= n; i++) par[i] = i;
    }
    int find(int u) {
        if (par[u] == u) return u;
        return par[u] = find(par[u]);
    }
    bool join(int u, int v) {
        u = find(u); v = find(v);
        if (u == v) return false;
        par[v] = u;
        return true;
    }
} dsu;

int n, m, totalWeight = 0; // n và m là số đỉnh và số cạnh
vector < Edge > edges;

int main() {
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);

    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v, c;
        cin >> u >> v >> c;
        edges.push_back({u, v, c});
    }

    dsu.init(n);

    sort(edges.begin(), edges.end(), [](Edge & x, Edge & y) {
        return x.c < y.c;
    });

    for (auto e : edges) {
        if (!dsu.join(e.u, e.v)) continue;

        totalWeight += e.c;
    }

    cout << totalWeight << '\n';
}
```

5. Các thuật toán tìm đường đi ngắn nhất

Thuật toán Dijkstra: Đồ thị trọng số dương, 1 to all, độ phức tạp O(V+E*logV).

```
const long long INF = 2000000000000000000LL;
struct Edge{// kiểu dữ liệu tự tạo để lưu thông số của một cạnh.
    Int v;
    long long w;
};
struct Node{// kiểu dữ liệu để lưu đỉnh u và độ dài của đường đi ngắn nhất từ s đến u.
    int u;
    long long Dist_u;
};
struct cmp{
    bool operator() (Node a, Node b) {
        return a.Dist_u > b.Dist_u;
    }
};
void dijkstraSparse(int n, int s, vector<vector<Edge>> &E, vector<long long> &D, vector<int> &trace) {
    D.resize(n, INF);
    trace.resize(n, -1);
    vector<bool> P(n, 0);

    D[s] = 0;
    priority_queue<Node, vector<Node>, cmp> h; // hàng đợi ưu tiên, sắp xếp theo dist[u] nhỏ nhất trước
    h.push({s, D[s]});

    while(!h.empty()) {
        Node x = h.top();
        h.pop();

        int u = x.u;
        if(P[u] == true) // Đỉnh u đã được chọn trước đó, bỏ qua
            continue;

        P[u] = true; // Đánh dấu đỉnh u đã được chọn
        for(auto e : E[u]) {
            int v = e.v;
            long long w = e.w;

            if(D[v] > D[u] + w) {
                D[v] = D[u] + w;
                h.push({v, D[v]});
                trace[v] = u;
            }
        }
    }
}
```

Thuật toán Bellman – Ford: Đồ thị có trọng số âm, 1 to all, độ phức tạp O(V*E).

```
const long long INF = 2000000000000000000LL;
struct Edge {
    int u, v;
    long long w; // cạnh từ u đến v, trọng số w
};
void bellmanFord(int n, int S, vector<Edge> &e, vector<long long> &D, vector<int> &trace) {
    // e: danh sách cạnh, n: số đỉnh, S: đỉnh bắt đầu, D: độ dài đường đi ngắn nhất
    // trace: mảng truy vết đường đi, INF nếu không có đường đi, -INF nếu có đường đi âm vô tận
    D.resize(n, INF);
    trace.resize(n, -1);
    D[S] = 0;
    for(int T = 1; T < n; T++) {
        for (auto E : e) {
            int u = E.u, v = E.v;
            long long w = E.w;
            if (D[u] != INF && D[v] > D[u] + w) {
                D[v] = D[u] + w;
                trace[v] = u;
            }
        }
    }
}
```

Thuật toán Floyd-Warshall: Có thể có trọng số âm, all to all, độ phức tạp O(V^3).

```
void floydWarshall(int n, vector<vector<long long>> &w, vector<vector<long long>> &D, vector<vector<int>>
&trace) {
    // w là ma trận kề, nếu không có cạnh nối thì w[u][v]=INF.
    D = w;
    init_trace(trace); // nếu cần dò đường đi, set trace[u][v]=u với mọi u,v

    for (int k = 0; k < n; k++) {
        for (int u = 0; u < n; u++) {
            for (int v = 0; v < n; v++) {
                if (D[u][v] > D[u][k] + D[k][v]) {
                    D[u][v] = D[u][k] + D[k][v];
                    D[u][v] = max(D[u][v], -INF);
                    trace[u][v] = trace[k][v];
                }
            }
        }
    }
}
```

Lưu ý: Floyd-warshall không tìm được chu trình âm, còn Bellman-Ford thì có thể. Với đồ thị thưa, không có trọng số âm, thay vì sử dụng thuật toán Floyd, ta có thể chạy thuật toán Dijkstra cải tiến N lần với N đỉnh nguồn để tìm đường đi ngắn nhất giữa mọi cặp đỉnh, với độ phức tạp tốt hơn thuật toán Floyd (nếu $E < V^2 / \log V$).

6. Luồng cực đại trên mạng

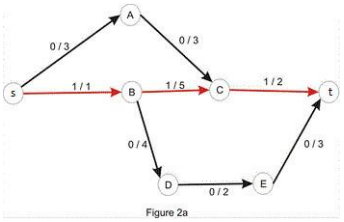


Figure 2a

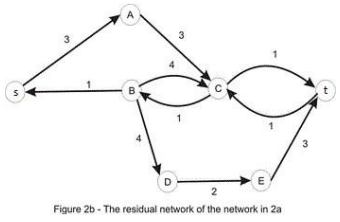


Figure 2b - The residual network of the network in 2a

Mạng thặng dư: Mạng thặng dư G′(E′,V′) của mạng G(E,V) cho biết sức chứa còn lại trên mạng G(E,V) khi đã gửi một số luồng f∗ qua nó và được xây dựng như sau:

- Tập đỉnh V'=V
- Mỗi cạnh e(u,v) ∈ E có giá trị luồng là $f[u,v]$ và sức chứa c[u,v] tương ứng với 2 cạnh trong E':

e' (u,v) (cạnh xuôi) có $f'[u,v] = c[u,v] - f[u,v]$, và e'(v,u) (cạnh ngược) có $f'[u,v] = f[u,v]$.

Đường tăng luồng: đường tăng luồng là một đường đi đơn từ đỉnh phát s (source) đến đỉnh thu t (sink) trong mạng thặng dư G′ mà kênh trên đường đi chưa bị bão hòa.

Thuật toán Ford-Fulkerson:

Bước 1: Tạo mạng thặng dư G'

Bước 2: Tìm một đường tăng luồng trên G', nếu tìm được thì thực hiện tăng luồng, không thì kết thúc. Ta có thể dùng DFS hoặc BFS để tìm đường tăng luồng và độ phức tạp là $O(nf.E)$ với nf là số lần tăng luồng. Nếu sử dụng PFS – Priority First Search thì độ phức tạp sẽ là O(E.log U) với U là khả năng thông qua lớn nhất.

Code:

```
int ml[1001][1001];  // dennote f[u,v] in G'
vector<int> trace(1001,-1), mfl(1001,0), edg[1001];
vector<bool> vis(1001,false);

class cmp{
public:
    bool operator() (const int &a,const int &b) const { return mfl[a]>mfl[b]; }
};
multiset<int, cmp> st;

int n,m,s,t, ans=0;

void pfs() {
    while(!st.empty()) {
        int node=*st.begin();
        st.erase(st.begin());
        if(vis[node]) continue;
        else vis[node]=true;
        if(node==t) return;
```

```
        for(auto it: edg[node]) {
            if(!vis[it] && ml[node][it]>0 && mfl[it]<min(mfl[node],ml[node][it]) ) {
                trace[it]=node;
                mfl[it]=min(mfl[node],ml[node][it]);
                st.insert(it);
            }
        }
    }
}

bool findArgument() {
    for_(n+1) { trace[i]=-1; vis[i]=false; mfl[i]=0; }
    mfl[s]=1e9;
    st.clear();
    st.insert(s);
    pfs();
    if(trace[t]==-1) return false;
    else return true;
}

void max_flow() {
    int minFlow=1e9;
    while(findArgument()) {
        int i=t;
        while(trace[i]!=-1) {
            minFlow=min(minFlow, ml[trace[i]][i]);
            i=trace[i];
        }
        i=t;
        while(trace[i]!=-1) {
            int j=trace[i];
            ml[j][i]-=minFlow;
            ml[i][j]+=minFlow;
            i=trace[i];
        }
        ans+=minFlow;
    }
}
```

Các bài toán liên quan:

Tìm số cặp ghép cực đại trên đồ thị phân đôi: Cho đồ thị phân đôi G gồm hai tập đỉnh phân đôi là A và B, các cạnh nối đỉnh từ A vào B. Loại bỏ một số cạnh sao cho sau khi thực hiện, tất cả mọi đỉnh của cạnh chỉ thuộc về nhiều nhất một cạnh và số cạnh còn lại là lớn nhất.

Bài này có thể đưa về tìm max flow bằng cách tạo thêm một đỉnh source và nối source tới tất cả đỉnh ở A, tạo thêm một đỉnh sink và nối tất cả đỉnh ở B tới sink. Set sức chứa của mỗi cạnh là 1. Khi đó số cặp ghép tối đa chính là giá trị luồng cực đại.

Mạng với khả năng thông qua của đỉnh và cạnh: Nâng dữ kiện của bài toán max flow thành: mỗi đỉnh cũng có một sức chứa tối đa cho phép như cạnh. Để giải quyết thì chỉ cần biến đỉnh thành cạnh. Giả sử có cạnh (2,1), (3,1) và (1,5), (1,6) và đỉnh 1 có sức chứa 2. Chỉ cần thay thành (2,1a), (3,1a), (1a,1b), (1b,5), (1b,6) với cạnh (1a,1b) có sức chứa 2.

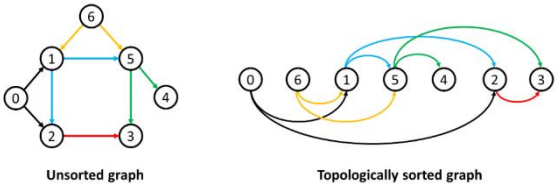
Lát cắt hẹp nhất: Cho đồ thị G có trọng số dương. Loại bỏ một số cạnh sao cho từ đỉnh s không thể tới được đỉnh t và số cạnh bị loại có tổng trọng số ít nhất.

Max-flow and min-cut theorem: Trong một mạng, giá trị luồng cực đại bằng với giá trị lát cắt cực tiểu.

Để giải quyết bài này, chỉ dùng giải thuật Ford-Fulkerson, sau đó xem trong mạng thặng dư, tìm tập tất cả những đỉnh có thể tới được từ source. Những cạnh nối từ tập này ra ngoài chính là các cạnh phải cắt.

7 . Sắp xếp topo

Sắp xếp Tô-pô được áp dụng nhiều nhất đối với các bài toán biểu diễn mối quan hệ phụ thuộc giữa các đối tượng. Một đồ thị tồn tại thứ tự Tô-pô khi và chỉ khi đồ thị đó phải là đồ thị có hướng và không có chu trình (Directed Acyclic Graph - DAG). Thứ tự Tô-pô không nhất thiết phải là duy nhất. Có thể có một số thứ tự Tô-pô khác nhau trong một đồ thị. Tuy nhiên, thứ tự Tô-pô sẽ là duy nhất khi DAG có đường đi Hamiltom.



Problem: Cho đồ thị có hướng không chu trình (DAG). Hãy đánh số lại G sao cho chỉ có cung nối từ đỉnh có chỉ số nhỏ đến chỉ số lớn.

Hướng giải sử sụng DFS:

- Xuất phát từ một điểm chưa được duyệt, ta bắt đầu duyệt DFS trên đồ thị xuất phát từ điểm đó.
- Dùng một mảng để lưu trạng thái của mỗi đỉnh. Có 3 trạng thái:
  - Trạng thái 0 : Đỉnh chưa được duyệt (chưa từng được gọi hàm DFS).
  - Trạng thái 1 : Đỉnh vẫn đang duyệt (hàm DFS với đỉnh này chưa kết thúc).
  - Trạng thái 2 : Đỉnh đã duyệt xong (hàm DFS với đỉnh này đã kết thúc).
- Hiển nhiên, khi đang duyệt mà ta gặp phải một đỉnh ở trạng thái 1 thì điều đó chứng tỏ đồ thị đang xét có chứa chu trình, và không thể sắp xếp Tô-pô được.
- Sau khi kết thúc duyệt DFS trên đồ thị, thứ tự hoàn tất duyệt của mỗi đỉnh chính là danh sách nghịch đảo thứ tự Tô-pô. Để có được thứ tự Tô-pô, đơn giản ta chỉ cần đảo ngược lại danh sách nghịch đảo thứ tự Tô-pô.

Code:

```
#include <bits/stdc++.h>
using namespace std;

const int maxN = 110;

int n, m;
int visit[maxN], ans[maxN];
vector <int> g[maxN];
stack <int> topo;

void dfs(int u) {
    visit[u] = 1;
    for (auto v : g[u]) {
        if (visit[v] == 1) {
            cout << "Error: graph contains a cycle";
            exit(0);
        }
        if (!visit[v]) dfs(v);
    }
    topo.push(u);
    visit[u] = 2;
}

main() {
    cin >> n >> m;
    while (m--) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
    }
    for (int i = 1; i <= n; ++i)
        if (!visit[i]) dfs(i);
    /* Sau khi xác định được thứ tự Tô-pô của đồ thị, ta sử dụng
       mảng ans để đánh số lại các đỉnh */
    int cnt = 0;
    while (!topo.empty()) {
        ans[topo.top()] = ++cnt;
        topo.pop();
    }

    for (int i = 1; i <= n; ++i) cout << ans[i] << ' ';
}
```

III. MỘT SỐ BÀI TOÁN QUY HOẠCH ĐỘNG.

1. Longest increasing subsequence

```
#include <bits/stdc++.h>
using namespace std;
int longest_increasing_subsequence(vector<int>& arr)
{
    vector<int> ans;
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        auto it
            = lower_bound(ans.begin(), ans.end(), arr[i]);
        if (it == ans.end()) {
            ans.push_back(arr[i]);
        }
        else {
            *it = arr[i];
        }
    }
    return ans.size();
}
```

2. 0-1 Knapsack

```
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    vector<vector<int> > K(n + 1, vector<int>(W + 1));

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}
```

3. Unbound Knapsack:

Có n đồ vật, vật thứ i có trọng lượng Ai và giá trị Bi. Hãy chọn ra một số các đồ vật để xếp vào vali có trọng lượng tối đa W sao cho tổng giá trị của vali là lớn nhất (Chú ý mỗi vật có thể chọn nhiều lần). Điều kiện: $1 \le n \times W \le 10^6, 1 \le Ai, Bi \le 10^9$.

Ta có thể gọi L[i][j] là giá trị lớn nhất có thể có khi ta chọn các vật từ 1 đến i sao cho khối lượng của chúng không vượt quá j. Khi đó:

$$\begin{cases} L[i][0] = L[0][j] = 0 \\ L[i][j] = L[i-1][j], if\ (A_i > j) \\ L[i][j] = \max(L[i-1][j], L[i][j - A_i] + B_i), else \end{cases}$$

Code:

```
int main()
{
    cin >> n >> w;
    a.resize(n + 1);
    b.resize(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> a[i] >> b[i];
```

```
    P = L = vector<long long>(w + 1);
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= w; j++)
        {
            if (a[i] > j)
                L[j] = P[j];
            else
                L[j] = max(P[j], L[j - a[i]] + b[i]);
        }
        P = L;
    }
    cout << L[w];
}
```

4 . SOS DP

```
// Sum over subset problem
// Define 1: 'i' is the subset of a binary number 'x' when  x & i = i
// Given an arrray a: a[1], a[2], ... a[n-1]
// F(x) = sum of all a[i] such that 'i' is the subset of x;
// Calculate F(0), F(1), ... F(n-1);

// Constraint: 1<=n<=10^6, 0<=a[i]<=10^9

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    //ifstream cin("input.txt");
    int n, N=20; // N is max number of bit
    cin >> n;
    vector<int> a(n); for_(n) cin >> a[i];
    vector<ll> sos(n); // Store F(x)
    vector<vector<int>> dp(1 << N, vector<int>(N+1)); // This what we call SOS DP

    // DP[mask][i] is the sum of all a[j] such that j is the subset of mask and just differ at most last
'i' bit.
    // For example: DP[10011011][3] = a[10011011] + a[10011010] + a[10011001] + a[10011000]
    // So to calculate DP[mask][x], we notice that if x_th bit is '0', so clearly that
DP[mask][x]=DP[mask][x-1] // like DP[10011][3]=DP[10011][2]=a[10011]+a[10010]+a[10001]+a[10000]
    // In case x_th bit is '1', DP[mask][x]=DP[mask][x-1]+DP[mask_with_x_th_bit_turns_0][x-1] or
DP[mask][x-1] + DP[mask ^ ( 1 << (x-1)) ][x-1]

    for( int mask=0; mask<n; mask++ ) {
        dp[mask][0]=a[mask];
        for(int i=1; i<=N; i++) {
            if( mask&( 1<<(i-1 )) ) dp[mask][i]=dp[mask][i-1]+dp[mask ^(1<<(i-1))][i-1];
            else dp[mask][i]=dp[mask][i-1];
        }
        sos[mask]=dp[mask][N];
    }

    cout << sos << el;
    return 0;
}
```

5. Xâu con chung dài nhất:

Cho 2 xâu X, Y. Hãy tìm xâu con của X và của Y có độ dài lớn nhất. Biết xâu con của một xâu thu được khi xóa một số kí tự thuộc xâu đó (hoặc không xóa kí tự nào).

Gọi L[i,j] là độ dài xâu con chung dài nhất của xâu X_i gồm i kí tự phần đầu của X và xâu Y_j gồm j kí tự phần đầu của Y. Ta có công thức quy hoạch động như sau:

- L[0,j] = L[i,0] = 0
- L[i,j] = L[i−1,j−1] + 1 nếu Xi=Yj
- L[i,j] = max(L[i−1,j], L[i,j−1]) nếu Xi≠Yj.

6 . Thêm ký tự để string là palidrome

Cho một xâu S, hãy tìm số kí tự ít nhất cần thêm vào S để S trở thành xâu đối xứng.

- Gọi L[i,j] là số kí tự ít nhất cần thêm vào xâu con S[i..j] của S để xâu đó trở thành đối xứng.
- Đáp số của bài toán sẽ là L[1,n] với n là số kí tự của S. Ta có công thức sau để tính L[i,j]:
  - L(i,i)=0.
  - L(i,j)=L(i+1,j−1) nếu Si=Sj
  - L(i,j)=max(L(i+1,j),L(i,j−1)) nếu Si≠Sj

IV. SỐ HỌC VÀ HÌNH HỌC

1. Kiểm tra số nguyên tố Miller-Rabin test:

```
ll pow_mod(ll x, ll y, ll mod) {
    ll ans=1;
    while(y>0) {
        if(y%2==1) ans=multiply_modulo(ans,x,mod);
        y=y>>1;
        x=multiply_modulo(x,x,mod);
    }
    return ans%mod;
}

bool witness(ll a, ll N) {
    ll k=0, m=N-1;
    while(m%2==0) {
        k++;
        m=m/2;
    }
    ll b=pow_mod(a,m,N);
    if(b==1 || b==N-1) return true;
    else {
        for(int i=1; i<k; i++) {
            b=multiply_modulo(b,b,N);
            if(b==N-1) return true;
        }
    }
    return false;
}

bool MillerRabinTest(ll N, ll a) {
    if(gcd(N,a)!=1) { return false; }
    if(pow_mod(a,N-1,N)!=1) { return false; }
    return witness(a,N);
}

bool testPrime(ll N, vector<bool> &isP) {
    if(N<1e6) return isP[N];
    ll r=rand()%500+100;
    for(int i=0; i<20; i++)
        if(!MillerRabinTest(N,i+r)) return false;
    return true;
}
```

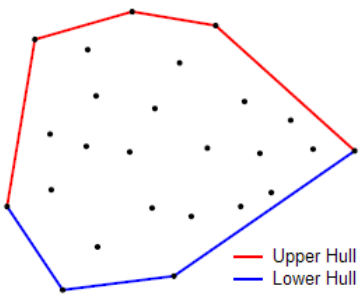
2. Inverse modulo

Inverse modulo m của a tồn tại nếu $(a,m)=1$.

$a^{\phi(m)-1} \equiv a^{-1} (\mathrm{mod}\, m)$. Nếu m là số nguyên tố thì $a^{m-2} \equiv a^{-1} (\mathrm{mod}\, m)$.

3. Bao lồi

Trong **hình học tính toán (computational geometry)**, **bao lồi (convex hull)** của một tập điểm là tập lồi (convex set) **nhỏ nhất** (theo diện tích, thể tích, …) mà tất cả các điểm đều nằm trong tập đó.



Thuật toán chuỗi đơn điệu

Sắp xếp tập điểm theo hoành độ, tung độ. Ta được điểm trái nhất và điểm phải nhất. Sau đi đi tìm bao lồi phía trên theo chiều kim đồng hồ. Lần lượt đi từ trái qua và thêm từ điểm phảo bao lồi H, kiểm tra xem 3 điểm gần nhất có bị "lõm" hay không, nếu có thì xóa điểm giữa gây lõm và xém tiếp 3 điểm cuối đến khi thỏa mãn.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

// Kiểu điểm
struct Point {
    int x, y;
};

// A -> B -> C đi theo thứ tự ngược chiều kim đồng hồ
bool ccw(const Point &A, const Point &B, const Point &C) {
    return 1LL * (B.x - A.x) * (C.y - A.y) - 1LL * (C.x - A.x) * (B.y - A.y) > 0;
}

// Trả về bao lồi với thứ tự các điểm được liệt kê theo chiều kim đồng hồ
vector<Point> convexHull(vector<Point> p, int n) {
    // Sắp xếp các điểm theo tọa độ x, nếu bằng nhau sắp xếp theo y
    sort(p.begin(), p.end(), [](const Point &A, const Point &B) {
        if (A.x != B.x) return A.x < B.x;
        return A.y < B.y;
    });

    // Tập bao lồi
    vector<Point> hull;
    hull.push_back(p[0]);

    // Dựng bao trên
    for (int i = 1; i < n; ++i) {
        while (hull.size() >= 2 && ccw(hull[hull.size() - 2], hull.back(), p[i])) {
            hull.pop_back();
        }
        hull.push_back(p[i]);
    }

    // Tiếp tục dựng bao dưới
    for (int i = n - 2; i >= 0; --i) {
        while (hull.size() >= 2 && ccw(hull[hull.size() - 2], hull.back(), p[i])) {
            hull.pop_back();
        }
        hull.push_back(p[i]);
    }

    // Xoá điểm đầu được lặp lại ở cuối
    if (n > 1) hull.pop_back();

    return hull;
}
```

V. TEAM'S TEMPLATE AND LIBRARY

1. Library

```cpp
#include <bits/stdc++.h> // for all

// if there is no <bits/stdc++.h>
#include <iostream>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <climits>
#include <stack>
#include <queue>
#include <vector>
#include <set>
#include <map>
#include <list>
#include <cassert>
#include <unordered_map>
```

2. Template for coding and debugging.

#1:

```
#define ll          long long
#define ull         unsigned long long
#define for_(n)     for(ll i=0; i<n; i++)
#define for__(a,b)  for(ll i=a; i<b; i++)
#define _for(i,a,b) for(ll i=a; i<b; i++)
#define mp          make_pair
#define fi          first
#define se          second
#define pb          push_back
#define pii         pair<int, int>
#define pll         pair<long long, long long>
#define el          "\n"
#define debug(x)    cerr << "[debug] " << #x << " : " << x << endl;

const long long MOD=1000000007;

template<class T, class P>
ostream& operator<<(ostream& os, const pair<T,P> &a) { os << "{" << a.first << ";" << a.second << "}";
return os; }
template<class T>
ostream& operator<<(ostream& os, const vector<T> &a) { ; for(auto it: a) os << it << " "; ; return os; }
template<class T>
ostream& operator<<(ostream& os, const deque<T> &a) { ; for(auto it: a) os << it << " "; ; return os; }
template<class T>
ostream& operator<<(ostream& os, const set<T> &a) { ; for(auto it: a) os << it << " "; ; return os; }
template<class T>
ostream& operator<<(ostream& os, const multiset<T> &a) { ; for(auto it: a) os << it << " "; ; return os; }

ll gcd(ll a, ll b) { return b==0? a : gcd(b,a%b); }
ll lcm(ll a, ll b) { return a/(gcd(a,b))*b; }

ll pow_mod(ll x, ll y, ll mod) { //mod<3.10^9
    ll ans=1;
    while(y>0) {
        if(y%2==1) ans=ans*x%mod;
        y=y>>1;
        x=x*x%mod;
    }
    return ans%mod;
}
```