比赛资料

Edited by chaoweic

预处理片段

```
#include <bits/stdc++.h>
#define fi first
#define se second
#define all(a) a.begin(),a.end()
#define rall(a) a.rbegin(),a.rend()
#define req(i,a,n) for(int i = a; i <= n; ++i)
#define rep(i,a,n) for(int i = a; i >= n; --i)
#define pb push_back
#define int long long
using namespace std;
using LL = long long;
typedef pair<int,int> PII;
typedef pair<long long,long long> PLL;
const int INF = 0x3f3f3f3f;
const LL mod = 1e9 + 7;
const int N = 1e5;
void Solve()
{}
signed main()
{
    ios::sync_with_stdio(false);
    cin.tie(0),cout.tie(0);
    Solve();
    return 0;
}
```

基础算法

排序

```
void quick_sort(int q[], int l, int r)
{
    //递归的终止情况
    if (l >= r) return;
    //选取分界线。这里选数组中间那个数
    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    //划分成左右两个部分
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    //对左右部分排序
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
void merge_sort(int q[], int l, int r)
{
    //递归的终止情况
    if (l >= r) return;
    //第一步：分成子问题
    int mid = l + r >> 1;
    //第二步：递归处理子问题(可以求逆序对)
    LL res = merge_sort(q, l, mid) + merge_sort(q, mid + 1, r);
    //第三步：合并子问题
    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else res += mid - i + 1; tmp[k ++ ] = q[j ++ ];
    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];
    //第四步：复制回原数组
    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

二分

```
//第一个大于等于x的数
int bsearch(int l,int r)
{
    int x;
    cin >> x;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (q[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
}
//找最后一个小于等于x的数
int bsearch(int l,int r)
{
    int x;
    cin >> x;
    while (l < r)
    {
        int mid = l + r +1  >> 1;
        if (q[mid] <= x) l = mid;
        else r = mid - 1;
    }
    return l;
}
//浮点二分
while(r-l>1e5)
{
    double mid = (l+r)/2;//注意不能用移位
    if(check(mid)) l=mid;
    else r=mid;
}
double y;//解方程
cin>>y;
double l = 0,r = 100;
for(int i=0;i<50;i++)
{
    double m = (l + r)/2;
    if(2018*m*m*m*m+21*m+5*m*m*m+5*m*m+14 >= y) r = m;
    else l = m;
```

```
}
if(l==0||r==100) cout<<-1<<endl;
else cout<<fixed<<setprecision(4)<<r<<endl;
//蓝桥杯分巧克力
bool check(int l) //判断一共能否分k块边长为l的巧克力
{
        int s = 0;
        for (int i = 1; i <= n; i++) s += (a[i] / l) * (b[i] / l);
        if (s >= k) return true;
        return false;
}
int main()
{
        cin >> n >> k;
        for (int i = 1; i <= n; i++)
        {
                cin >> a[i] >> b[i];
                m = max(max(a[i], b[i]), m);//最大边长
        }
        int mid, l = 0, r = m + 1;//二分查找 初始化边长区间
        while (l + 1 < r)
        {
                mid = (l + r )/2;
                if (check(mid)) l = mid;//尝试更长边长
                else r = mid;//更小边长
        }
        if (check(l)) cout << l << endl;
        else cout << r << endl;
}
```

高精度

```
vector<int> add(vector<int> &A, vector<int> &B) //加法
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    req(i,0,A.size() - 1)
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }
    if (t) C.push_back(t);
    return C;
}
void mutiply() //乘法
{
    char a1[10001], b1[10001];
    int a[10001], b[10001], i, x, len, j, c[10001];
    int lena = strlen(a1);
    int lenb = strlen(b1);
    req(i,1,lena)
        a[i] = a1[lena - i] - '0';
    req(i,1,lenb)
        b[i] = b1[lenb - i] - '0';
    req(i,1,lenb)
        req(j,1,lena)
            c[i + j - 1] += a[j] * b[i];
    req(i,1,lena + lenb - 1)
        if (c[i] > 9)
        {
            c[i + 1] += c[i] / 10;
            c[i] %= 10;
        }
    len = lena + lenb;
    while (c[len] == 0 && len > 1)
        len--;
    rep(i,len,1)
```

```
        cout << c[i];
}
```

前缀和与差分

```
for (int i = 1; i <= n; i ++ ) s[i] = s[i - 1] + q[i]; //一维前缀和
s[r]-s[l-1];//下标l-1到r所有数的和，本质预处理操作
s[i][j] += s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1];//构造二维前缀和
s[x2][y2] - s[x1 - 1][y2] - s[x2][y1 - 1] + s[x1 - 1][y1 - 1] //查询操作
b[i] = a[i] - a[i - 1];      //构建差分数组
b[l] += c; b[r + 1] -= c;//原数组l到r全部加c
b[i] += b[i - 1];  //复原原数组
void insert(int x1, int y1, int x2, int y2, int c) //二维差分
{
    b[x1][y1] += c;
    b[x2 + 1][y1] -= c;
    b[x1][y2 + 1] -= c;
    b[x2 + 1][y2 + 1] += c;
}
req(i,1,n)
        req(j,1,m)
            insert(i, j, i, j, a[i][j]); //初始化差分数组b
insert(x1, y1, x2, y2, c); //原数组局部矩阵全部加c
b[i][j] += b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1];//复原
//线段覆盖
vector<int> a(n),b(n),c(n);
req(i,0,n-1)
    cin >> a[i];
req(i,1,n-1)
   b[i] = a[i] - a[i - 1];
sort(b.begin(), b.end());
req(i, 1, n - 1)
    c[i] = c[i-1] + b[i];
for (int i = n-1; i >= 0;i--)
    cout << c[i] << " ";
```

双指针

```
int res = 0; //最长不重复子序列的长度
for (int i = 0, j = 0; i < n; i ++ )
{
    s[q[i]] ++ ;//q是原数组 s用来记录区间内元素的数量
    while (j < i && s[q[i]] > 1) s[q[j ++ ]] -- ;
    res = max(res, i - j + 1);
}
for (int i = 0, j = m - 1; i < n; i ++ )//数组元素的目标和
{
    while (j >= 0 && a[i] + b[j] > x) j -- ;
    if (j >= 0 && a[i] + b[j] == x) cout << i << ' ' << j << endl;
}
int i = 0, j = 0;//判断子序列
while (i < n && j < m)
{
    if (a[i] == b[j]) i ++ ;
    j ++ ;
}
```

离散化

```
vector<int> alls;//存入下标容器
vector<PII> add, query;//add增加容器，存入对应下标和增加的值的大小
//query存入需要计算下标区间和的容器
int find(int x)
{
    int l = 0, r = alls.size() - 1;
    while (l < r)//查找大于等于x的最小的值的下标
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1;//因为使用前缀和，其下标要+1可以不考虑边界问题
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < n; i ++ )
    {
        int x, c;
        cin >> x >> c;
        add.push_back({x, c});//存入下标即对应的数值c

        alls.push_back(x);//存入数组下标x=add.first
    }

    for (int i = 0; i < m; i ++ )
    {
        int l, r;
        cin >> l >> r;
        query.push_back({l, r});//存入要求的区间

        alls.push_back(l);//存入区间左右下标
        alls.push_back(r);
    }

    // 区间去重
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(), alls.end()), alls.end());
```

```
    // 处理插入
    for (auto item : add)
    {
        int x = find(item.first);//将add容器的add.secend值存入数组a[]当中，
        a[x] += item.second;//在去重之后的下标集合alls内寻找对应的下标并添加数值
    }

    // 预处理前缀和
    for (int i = 1; i <= alls.size(); i ++ ) s[i] = s[i - 1] + a[i];

    // 处理询问
    for (auto item : query)
    {
        int l = find(item.first), r = find(item.second);
        cout << s[r] - s[l - 1] << endl;
    }
}
//在下标容器中查找对应的左右两端[l~r]下标，然后通过下标得到前缀和相减再得到区间a[l~r]的和
```

区间合并

```
vector<pair<int, int>> intervals, result;
static bool cmp(pair<int, int> &a, pair<int, int> &b) {
    return a.first < b.first;
}
int main () {
    cin >> n;
    for (int i = 0; i < n; i++) {
        int l, r;
        cin >> l >> r;
        intervals.push_back ({l, r});
    }
    sort(intervals.begin(), intervals.end(), cmp);

    result.push_back(intervals[0]);  // 将第一个区间加入

    for (int i = 1; i < intervals.size(); i++) {
        if (result.back().second >= intervals[i].first) {
            result.back().second = max(result.back().second, intervals[i].second);
        } else {
            result.push_back (intervals[i]);
        }
    }
    cout << result.size() << endl;//合并后区间个数
}
```

数据结构

链表

```
// head 表示头结点的下标 e[i] 表示节点i的值 ne[i] 表示节点i的next指针是多少
// idx 存储当前已经用到了哪个点 采用数组模拟链表
int head, e[N], ne[N], idx;
void init()
{
    head = -1;
    idx = 0;
}
void add_to_head(int x) // 将x插到头结点
{
    e[idx] = x, ne[idx] = head, head = idx ++ ;
}
void add(int k, int x) // 将x插到下标是k的点后面
{
    e[idx] = x, ne[idx] = ne[k], ne[k] = idx ++ ;
}
void remove(int k) // 将下标是k的点后面的点删掉
{
    ne[k] = ne[ne[k]];
}
```

栈

```
stack<type> a; .push(x) .pop() //移除栈顶元素
.top() .empty() .size()
```

队列

```
queue<type> q; .front() .back()
.push() .pop()
.size() .empty()
deque<type> dq;//双端队列 可以双向进出
.pop_back() .pop_front() //push同理
```

堆

```
priority_queue<type,vector<type>,greater<type>> pq; .top()//默认大根堆
.push(x) .pop() // 队顶元素出队
.size() .empty()
```

并查集

```
struct DSU {
    vector<int> p, r;
    DSU(int n=0){init(n);}
    void init(int n){ p.resize(n); iota(p.begin(),p.end(),0); r.assign(n,0); }
    int find(int x){ return p[x]==x?x:p[x]=find(p[x]); }
    bool unite(int a,int b){
        a=find(a); b=find(b);
        if(a==b) return false;
        if(r[a]<r[b]) swap(a,b);
        p[b]=a;
        if(r[a]==r[b]) r[a]++;
        return true;
    }
};
```

树状数组

```
struct BIT {
    int n; vector<LL> t;
    BIT(int _n=0){init(_n);}
    void init(int _n){ n=_n; t.assign(n+1,0); }
    void add(int i, ll v){ for(; i<=n; i+=i&-i) t[i]+=v; }
    LL sum(int i){ ll s=0; for(; i>0; i-=i&-i) s+=t[i]; return s; }
    LL sum(int l, int r){ return sum(r)-sum(l-1); }
};
```

线段树

```
struct Seg {
    int n; struct Node{ ll sum, add; } ;
    vector<Node> t;
    Seg(int _n=0){init(_n);}
    void init(int _n){ n=_n; t.assign(4*n+4,{0,0}); }
    void pull(int o){ t[o].sum = t[o<<1].sum + t[o<<1|1].sum; }
    void apply(int o, int l, int r, ll v){
        t[o].sum += v * (r-l+1);
        t[o].add += v;
    }
    void push(int o, int l, int r){
        if(t[o].add!=0){
            int m=(l+r)>>1;
            apply(o<<1, l, m, t[o].add);
            apply(o<<1|1, m+1, r, t[o].add);
            t[o].add=0;
        }
    }
    void add(int o,int l,int r,int L,int R,ll v){
        if(L>r||R<l) return;
        if(L<=l&&r<=R){ apply(o,l,r,v); return; }
        push(o,l,r);
        int m=(l+r)>>1;
        add(o<<1,l,m,L,R,v); add(o<<1|1,m+1,r,L,R,v);
        pull(o);
    }
    ll query(int o,int l,int r,int L,int R){
        if(L>r||R<l) return 0;
        if(L<=l&&r<=R) return t[o].sum;
        push(o,l,r);
        int m=(l+r)>>1;
        return query(o<<1,l,m,L,R)+query(o<<1|1,m+1,r,L,R);
    }
    // wrapper: add(1,1,n,L,R,v), query(1,1,n,L,R)
};
```

单调队列

```
vector<int> slide_max(const vector<int>& a, int k){
    deque<int> dq; vector<int> res;
    req(i,0,a.size()-1)
    {
        while(!dq.empty() && a[dq.back()]<=a[i]) dq.pop_back();
        dq.push_back(i);
        if(dq.front() <= i-k) dq.pop_front();
        if(i>=k-1) res.push_back(a[dq.front()]);
    }
    return res; //滑动窗口max/min
}
```

Trie

```
struct BinTrie //二进制字典树
{
    struct Node{ int ch[2]; Node(){ch[0]=ch[1]=-1;} };
    vector<Node> t;
    BinTrie(){ t.push_back(Node()); }
    void insert(int x, int B=30){
        int u=0;
        rep(i,B,0)
        {
            int b=(x>>i)&1;
            if(t[u].ch[b]==-1){ t[u].ch[b]=t.size(); t.push_back(Node()); }
            u=t[u].ch[b];
        }
    }
    int max_xor(int x, int B=30){
        int u=0, ans=0;
        rep(i,B,0)
        {
            int b=(x>>i)&1;
            int want = b^1;
            if(t[u].ch[want]!=-1){ ans |= (1<<i); u=t[u].ch[want]; }
            else u = t[u].ch[b];
        }
        return ans;
    }
};
```

哈希表

```
struct FastHash {
    size_t operator()(uint64_t x) const {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
};
// unordered_map<long long, int, FastHash> mp;
//map和unordered_map的区别：前者遍历排序时按字典序升序排，后者无序。并且前者O(logn)后者理想情况下O(1
unordered_set<type> ut;//无序集合
.size() .empty() .find() .count()//出现次数
insert() erase()
```

ST表

```cpp
template<typename T, typename F>
struct SparseTable {
    int n, LG;
    vector<vector<T>> st;
    vector<int> lg;
    F combine;
    SparseTable() {}
    SparseTable(const vector<T>& a, F f): combine(f) { build(a); }
    void build(const vector<T>& a){
        n = a.size();
        if(n==0) return;
        LG = 32 - __builtin_clz(n);
        st.assign(LG, vector<T>(n));
        lg.assign(n+1,0);
        for(int i=2;i<=n;i++) lg[i] = lg[i>>1] + 1;
        st[0] = a;
        for(int k=1;k<LG;k++){
            int len = 1<<k;
            for(int i=0;i+len-1<n;i++){
                st[k][i] = combine(st[k-1][i], st[k-1][i + (1<<(k-1))]);
            }
        }
    }
    T query(int l,int r){
        int k = lg[r-l+1];
        return combine(st[k][l], st[k][r-(1<<k)+1]);
    }
};
// min
SparseTable<int, function<int(int,int)>> st(a, [](int x,int y){ return min(x,y); });
// max
SparseTable<int, function<int(int,int)>> st2(a, [](int x,int y){ return max(x,y); });
// gcd (需要 include <numeric>)
SparseTable<int, function<int(int,int)>> stg(a, [](int x,int y){ return std::gcd(x,y); });
```

搜索与图论

DFS

```
//递归形式
vector<vector<int>> g;
vector<char> vis;
void dfs(int u){
    vis[u] = 1;
    // process u
    for(int v: g[u]){
        if(!vis[v]) dfs(v);
    }
}
//非递归形式
vector<int> st;
st.push_back(start);
vis[start]=1;
while(!st.empty()){
    int u = st.back(); st.pop_back();
    // process u
    for(int v: g[u]){
        if(!vis[v]){
            vis[v]=1;
            st.push_back(v);
        }
    }
}
//八皇后
int print()
{
    if(tol<=2)
    {
        req(k,1,n)
        {
            cout << a[k] << " ";
            cout << endl;
        }
    }
    tol++;
}
void dfs(int i)
```

```
{
    if(i>n)
    {
        print();
        return;
    }
    else
    {
        req(j,1,n)
        {
            if((!b[j])&&(!c[i+j])&&(!d[i-j+n]))
            {
                a[i] = j;b[j] = 1;c[i + j] = 1;d[i - j + n] = 1;
                dfs(i + 1);
                b[j] = 0;c[i + j] = 0;d[i - j + n] = 0;//回溯
            }
        }
    }
}
//回溯/组合/全排列(带剪枝)
vector<int> sol;
vector<char> used(n,false);
void backtrack(int pos){
    if(pos == target_len){
        // handle sol
        return;
    }
    for(int i=0;i<n;i++){
        if(used[i]) continue; // 剪枝示例：如果放 i 导致不可能达到最优，continue;
        used[i]=1; sol.push_back(i);
        backtrack(pos+1);
        sol.pop_back(); used[i]=0;
    }
}
```

BFS

```
//无权最短路
vector<int> dist(n, -1), par(n, -1);
queue<int> q;
dist[s]=0; q.push(s);
while(!q.empty()){
    int u=q.front(); q.pop();
    for(int v: g[u]){
        if(dist[v]==-1){
            dist[v]=dist[u]+1;
            par[v]=u;
            q.push(v);
        }
    }
}
//迷宫最短路
queue<PII> q;
    memset(d, -1, sizeof d);
    d[0][m-1] = 0;
    q.push({0, m-1});
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    while (q.size())
    {
        auto t = q.front();
        q.pop();
        for (int i = 0; i < 4; i++)
        {
            int x = t.first + dx[i], y = t.second + dy[i];
            if (x >= 0 && x < n + 2 && y >= 0 && y < m && p[x][y] && d[x][y] == -1)
            {
                d[x][y] = d[t.first][t.second] + 1;
                q.push({x, y});
                if(x == n+1 && y == 0)
                {
                    return d[x][y];
                }
            }
        }
    }
    queue<int> qa, qb;

//双向bfs
```

```
vector<int> da(n,-1), db(n,-1);
qa.push(sa); da[sa]=0;
qb.push(sb); db[sb]=0;
int ans = -1;
while(!qa.empty() && !qb.empty()){
    // expand one side (choose smaller frontier)
    if(qa.size() <= qb.size()){
        int u=qa.front(); qa.pop();
        for(int v: g[u]){
            if(da[v]==-1){ da[v]=da[u]+1; qa.push(v);
            if(db[v]!=-1){ ans = da[v]+db[v]; break; } }
        }
    } else {
        int u=qb.front(); qb.pop();
        for(int v: g[u]){
            if(db[v]==-1){ db[v]=db[u]+1; qb.push(v);
            if(da[v]!=-1){ ans = da[v]+db[v]; break; } }
        }
    }
    if(ans!=-1) break;
}
```

记忆化搜索

```
//滑雪
int dx[4] = {0, 0, 1, -1};
int dy[4] = {1, -1, 0, 0};
int n, m, a[201][201], s[201][201], ans;
bool use[201][201]; // 这个就是所谓的不需要
int dfs(int x, int y)
{
    if (s[x][y])
        return s[x][y]; // 记忆化搜索
    s[x][y] = 1;        // 题目中答案是有包含这个点的
    for (int i = 0; i < 4; i++)
    {
        int xx = dx[i] + x;
        int yy = dy[i] + y; // 四个方向
        if (xx > 0 && yy > 0 && xx <= n && yy <= m && a[x][y] > a[xx][yy])
        {
            dfs(xx, yy);
            s[x][y] = max(s[x][y], s[xx][yy] + 1);
        }
    }
    return s[x][y];
}
```

Dijkstra

```
//带路径回溯求最短路
struct Edge //图结构
{
    int to;
    LL weight;
    Edge(int t, LL w) : to(t), weight(w) {}
};
vector<vector<Edge>> g(N);
vector<LL> dj(int start, int n, vector<int> &parent)
{
    vector<LL> dist(n + 1, INF);
    vector<bool> vis(n + 1, false);
    parent.assign(n + 1, -1);
    priority_queue<PII, vector<PII>, greater<PII>> pq;
    dist[start] = 0;
    pq.push({0, start});
    while (!pq.empty())
    {
        int u = pq.top().second;
        LL d = pq.top().first;
        pq.pop();
        if (vis[u])
            continue;
        vis[u] = true;
        for (const Edge &e : g[u])
        {
            int v = e.to;
            LL w = e.weight;
            if (dist[u] + w < dist[v])
            {
                dist[v] = dist[u] + w;
                parent[v] = u;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
void print(int start, int end, const vector<int> &parent) //打印路径
{
    if (parent[end] == -1 && end != start)
```

```
    {
        cout << -1 << endl;
        return;
    }
    vector<int> path;
    int cur = end;
    while (cur != -1)
    {
        path.push_back(cur);
        cur = parent[cur];
    }
    reverse(path.begin(), path.end());
    for (int i = 0; i < path.size(); i++)
    {
        cout << path[i];
        if (i < path.size() - 1)
            cout << " ";
    }
    cout << endl;
}
void add(int u, int v, int w) //建图
{
    g[u].push_back(Edge(v, w));
    g[v].push_back(Edge(u, w));
}
void Solve()
{
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w);
    }
    vector<int> parent;
    vector<LL> dist = dj(1, n, parent);
    if (dist[n] == INF)
    {
        cout << -1 << endl;
    }
    else
```

```
        print(1, n, parent);
}
```

SPFA

```
int spfa() //可以包含负权边求最短路
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

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

    while (q.size())
    {
        int t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return dist[n];
}
bool spfa() //判断是否存在负环
{
    queue<int> q;

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

```
    }

    while (q.size())
    {
        int t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;

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

最小生成树

```
struct Edge {
    int u, v, w;
    bool operator<(const Edge& other) const {
        return w < other.w;
    }
};

vector<Edge> edges;
int parent[N];

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

int kruskal(int n) {
    sort(edges.begin(), edges.end());
    for (int i = 1; i <= n; i++) parent[i] = i;
    int total_weight = 0, cnt = 0;
    for (const auto& e : edges) {
        int pu = find(e.u), pv = find(e.v);
        if (pu != pv) {
            parent[pu] = pv;
            total_weight += e.w;
            cnt++;
            if (cnt == n - 1) break;
        }
    }
    return total_weight;
}
```

动态规划

- 主要思路：
  1.状态表示f[i][j];
  2.状态转移方程(集合划分)

背包问题

```
req(i,1,n)
{
    rep(j,m,v[i])
    {
        f[j] = max(f[j], f[j - v[i]]+w[i]);//01背包
    }
}
req(i,1,n)
{
    req(j,0,m)
    {
        for (int k = 0; k <= s[i] && k * v[i] <= j;k++) //多重背包
    }
}
req(i,1,n)
{
    rep(j,m,0)
    {
        req(k,0,s[i]-1)
        {
            if(v[i][k]<=j)
                f[j] = max(f[j], f[j - v[i][k]] + w[i][k]); //分组背包
        }
    }
}
req(i,1,n)
{
    req(j,v[i],m)
    {
        f[j] = max(f[j], f[j - v[i]] + w[i]); //完全背包
    }
}
```

线性dp

```
//数字三角形
f[1][1] = a[1][1];
req(i,2,n)
{
    req(j,1,i)
    {
        f[i][j] = max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i][j]);
    }
}
int res = -INF;
req(i,1,n)
{
    res = max(res, f[n][i]);
}
//最长上升子序列
req(i,1,n)
{
    f[i] = 1;
    req(j,1,i-1)
    {
        if(a[j]<a[i])
        {
            f[i] = max(f[i], f[j] + 1);
        }
    }
}
int res = 0;
req(i, 1, n){
    res = max(res, f[i]);
}
//最长公共子序列
req(i,1,n)
{
    req(j,1,m)
    {
        f[i][j] = max(f[i - 1][j], f[i][j - 1]);
        if(a[i] == b[j])
            f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
    }
}
//最短编辑距离
```

```
req(i,0,m)
    f[0][i] = i;
req(i,0,n)
    f[i][0] = i;
req(i,1,n)
{
    req(j,1,m)
    {
        f[i][j] = min(f[i - 1][j] + 1, f[i][j - 1] + 1);
        if(a[i] == b[j])
            f[i][j] = min(f[i][j], f[i - 1][j - 1]);
        else
            f[i][j] = min(f[i][j], f[i - 1][j - 1] + 1);
    }
}
```

区间dp

```
//石子合并最小代价
for (int i = 1; i <= n; i++)
    s[i] += s[i - 1];
for (int len = 2; len <= n; len++) {
    for (int i = 1; i + len - 1 <= n; i++)
    {
        int l = i, r = i + len - 1;
        f[l][r] = 1e8;
        for (int k = l; k < r; k++)
            f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s[l - 1]);
    }
    printf("%d\n", f[1][n]);
}
```

其他dp模型

```
// 状态机DP
int dp[N][2]; // 0/1表示状态
dp[i][0] = max(dp[i-1][0], dp[i-1][1]);
dp[i][1] = dp[i-1][0] + w[i];

// 树形DP模板
void dfs(int u, int fa) {
    dp[u][0] = 0; // 不选u
    dp[u][1] = w[u]; // 选u
    for(int v : g[u]) {
        if(v == fa) continue;
        dfs(v, u);
        dp[u][0] += max(dp[v][0], dp[v][1]);
        dp[u][1] += dp[v][0];
    }
}

// 状压DP
for(int mask = 0; mask < (1<<n); mask++) {
    for(int i = 0; i < n; i++) {
        if(mask >> i & 1) {
            int pre = mask ^ (1 << i);
            dp[mask] = max(dp[mask], dp[pre] + ...);
        }
    }
}
```

数学理论

数论

```
//欧拉筛
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉
void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
//求x约数个数
int x;cin >> x;
for (int i = 2; i <= x / i; i ++ ) {
    while (x % i == 0)
    {
        x /= i;
        primes[i] ++ ;
    }

    if (x > 1) primes[x] ++ ;
}
//求n个数乘积的约数之和 (自然数分解原理)
unordered_map<int, int> primes;
while (n -- )
{
    int x;
    cin >> x;
    for (int i = 2; i <= x / i; i ++ )
        while (x % i == 0)
        {
            x /= i;
            primes[i] ++ ;
        }
```

```
    if (x > 1) primes[x] ++ ;
}
LL res = 1;
for (auto p : primes)
{
    LL a = p.first, b = p.second;
    LL t = 1;
    while (b -- ) t = (t * a + 1) % mod;
    res = res * t % mod;
}
//最大公约数
int gcd(int a, int b)
{
    return b ? gcd(b, a % b) : a;
}
```

快速幂

```
//返回pow(a,b) mod p
LL qm(int a, int b, int p)
{
    LL res = 1 % p;
    while (b)
    {
        if (b & 1) res = res * a % p;
        a = a * (LL)a % p;
        b >>= 1;
    }
    return res;
}
```

排列组合

```
//求组合数 n<=2000
void init()
{
    req(i,0,N-1)
        req(j,0,i)
            if (!j) c[i][j] = 1;
            else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;//预处理组合数数组
}
//容斥原理
//字符串缩写太多了
long long mod = 1000000007, n, ans = 0, k, i;
cin >> n;
req(i,0,n-1)
    scanf("%s", s);
k = 1;
req(i,n,1)
{
    k = (k * i) % mod;
    ans = (ans + k) % mod;//所有缩写个数
}
```

位运算

```
& | ^ //与或非
~ //取反
```

博弈论

关于公平组合游戏，公平组合游戏中，
最基础也最重要的是正常 Nim 游戏。Sprague–Grundy 定理指出，所有正常规则的公平组合游戏都等价于一个单堆 Nim 游戏。由此，可以发展出 Sprague–Grundy 函数和 Nim 数的概念，它们完全地刻画了一个正常规则的公平组合游戏。

博弈图：将每一个可能的状态都看作是图中的一个结点，并将状态向它的后继状态（即通过一次操作可以达到的状态）连边，就得到一个有向无环图，这就是博弈图。

```
//nim游戏：共有 n  堆石子，第 i 堆有 ai 枚石子。
//两名玩家轮流取走任意一堆中的任意多枚石子，但不能不取。取走最后一枚石子的玩家获胜。
req(i,1,n)
{
    cin >> a[i];
    cnt ^= a[i];//nim和
}
Nim 游戏中，状态 (a1,a2,⋯,an)是必败态当且仅当nim和为0
```

引理

1.正常规则的公平组合游戏中，没有后继状态的状态是必败状态 P

2.一个状态是必胜状态当且仅当至少一个它的后继状态为必败状态

3.一个状态是必败状态当且仅当它的所有后继状态均为必胜状态

SG（Grundy）函数定义：对状态 s，g(s)=mex{g(t) | t∈moves(s)}，其中 mex 为最小非负整数不在集合中。

Sprague–Grundy 定理：任意若干独立子博弈的合成博弈的等价筹码值是各子博弈 SG 值的异或。合成博弈先手必胜当且仅当⊕igi≠0。

字符串

字符串哈希

```
using ULL = unsigned long long;
const ULL base = 131;
ULL h[N], p[N];

void init_hash(const string& s) {
    p[0] = 1;
    for (int i = 1; i <= s.size(); i++) {
        h[i] = h[i-1] * base + s[i-1];
        p[i] = p[i-1] * base;
    }
}

ULL get_hash(int l, int r) {
    return h[r] - h[l-1] * p[r-l+1];
}
```

KMP

```
vector<int> Next;
void GetNext(const string& p) {
    int len = p.size();
    Next.resize(len);
    int j = 0, k = -1;
    Next[0] = -1;
    while (j < len - 1) {
        if (k == -1 || p[k] == p[j]) {
            j++;
            k++;
            Next[j] = k;
        }
        else {
            k = Next[k];
        }
    }
}
int KMP(const string& t, const string& p) {
    int n = t.size(), m = p.size();
    if (m == 0) return 0;
    GetNext(p);
    int i = 0, j = 0;
    while (i < n && j < m) {
        if (j == -1 || t[i] == p[j]) {
            i++;
            j++;
        }
        else {
            j = Next[j];
        }
    }
    if (j == m) {
        return i - j;
    }
    else {
        return -1;
    }
}
```

AC自动机

```
//简易版
struct Tree // 字典树
{
    int fail;    // 失配指针
    int vis[26]; // 子节点的位置
    int end;     // 标记有几个单词以这个节点结尾
} AC[1000000];   // Trie树
int cnt = 0;     // Trie的指针
inline void Build(string s)
{
    int l = s.length();
    int now = 0;                // 字典树的当前指针
    for (int i = 0; i < l; ++i) // 构造Trie树
    {
        if (AC[now].vis[s[i] - 'a'] == 0)    // Trie树没有这个子节点
            AC[now].vis[s[i] - 'a'] = ++cnt; // 构造出来
        now = AC[now].vis[s[i] - 'a'];       // 向下构造
    }
    AC[now].end += 1; // 标记单词结尾
}
void Get_fail() // 构造fail指针
{
    queue<int> Q;                // 队列
    for (int i = 0; i < 26; ++i) // 第二层的fail指针提前处理一下
    {
        if (AC[0].vis[i] != 0)
        {
            AC[AC[0].vis[i]].fail = 0; // 指向根节点
            Q.push(AC[0].vis[i]);      // 压入队列
        }
    }
    while (!Q.empty()) // BFS求fail指针
    {
        int u = Q.front();
        Q.pop();
        for (int i = 0; i < 26; ++i) // 枚举所有子节点
        {
            if (AC[u].vis[i] != 0) // 存在这个子节点
            {
                AC[AC[u].vis[i]].fail = AC[AC[u].fail].vis[i];
                // 子节点的fail指针指向当前节点的
```

```
                // fail指针所指向的节点的相同子节点
                Q.push(AC[u].vis[i]); // 压入队列
            }
            else // 不存在这个子节点
                AC[u].vis[i] = AC[AC[u].fail].vis[i];
            // 当前节点的这个子节点指向当
            // 前节点fail指针的这个子节点
        }
    }
}
int AC_Query(string s) // AC自动机匹配
{
    int l = s.length();
    int now = 0, ans = 0;
    for (int i = 0; i < l; ++i)
    {
        now = AC[now].vis[s[i] - 'a'];                          // 向下一层
        for (int t = now; t && AC[t].end != -1; t = AC[t].fail) // 循环求解
        {
            ans += AC[t].end;
            AC[t].end = -1;
        }
    }
    return ans;
}
req(i,1,n) {
    {
        cin >> s;
        Build(s);
    }
    AC[0].fail = 0; // 结束标志
    Get_fail();     // 求出失配指针
    cin >> s;       // 文本串
    cout << AC_Query(s) << endl;
}
```

STL

容器

这里只总结string

```
string b = a + s; //字符串拼接
getchar(); //cin.get()
getline(cin, s);//读入下一行的输入
const char *m = s.c_str() //string转换char数组
.size() .length()
.push_back(x)
.insert(pos,x)
.append(str) //在字符串结尾添加str字符串
erase(iterator) //删除字符串中所指的字符
erase(pos, len) //删除字符串从pos开始的len个字符
.clear()
.replace(pos,n,str) //把当前字符串从索引pos开始的n个字符替换为str
.replace(pos, n, n1, c) //把当前字符串从索引pos开始的n个字符替换为n1个字符c
tolower(s[i]) //转换小写
toupper(s[i]) //转换大写
.substr(pos,n) //截取从pos索引开始的n个字符
sort(s.begin(),s.end());  //按字符的ASCII码排序
reverse(s.begin(), s.end());//字符串反转
.find(str,pos) //在pos索引位置开始查找子串str,返回找到的位置索引
```

迭代器

```
vector<int>::iterator it;//定义vector的迭代器
it = a.begin();//让it指向a的第一个元素，相当于一个指针
it++;//向前移动
cout<<*it<<endl;//输出元素，2，*表示解引用
it = erase(x);//注意用迭代器进行删除操作时要对it重新赋值
it = insert(x) //同上，只建立用迭代器进行访问，而非操作元素
```

函数

```
accumulate(begin,end,intit); // 求和
atoi(const char *) //将字符串转换为int
fill(begin,end,num) //对一个序列初始化赋值
is_sorted(begin,end)//判断序列是否有序
lower_bound(a.begin,a.end())/upper_bound(a.begin,a.end()) //二分查找。前面找>x，后面找>=x
max_element()/min_element(a, a + n); //找最大最小值
max()/min()//多个元素最大最小值
minmax_element(begin, end) //返回序列中最小和最大值组成的pair的对应地址
nth_element(begin,nth,end) //寻找第序列第n小的值
next_permutation(begin, end) //求序列的下一个排列
partial_sort(begin, mid, end) //部分排序begin到mid
partial_sum(begin,end,back_inserter(a)) //将前缀和插入到a后面
random_shuffle(a,a+n) //打乱序列顺序
reverse(begin,end) //反转序列
sort(begin, end,cmp);
to_string(int a) //将数字转化为字符串
unique(begin,end) //去重，将重复数字移到序列最后面
_gcd(a,b) //求最大公约数
_lcm(a,b) //求最小公倍数
```

结构体详解

```
struct Point {
    int x;
    int y;
    // 默认 public
    // 构造函数
    Point(int _x=0, int _y=0): x(_x), y(_y) {}

    // 成员函数（常成员）
    double norm() const { return sqrt((double)x*x + (double)y*y); }

    // 运算符重载（比较，用于 sort）
    bool operator<(Point const& other) const {
        if(x != other.x) return x < other.x;
        return y < other.y;
    }
};
//STL使用
vector<Point> v;
v.emplace_back(1,2); // 直接构造，避免拷贝
sort(v.begin(), v.end()); // 需 operator< 或 comparator

//常用模板
//排序和去重
struct Item { int a,b; };
vector<Item> a = ...;
sort(a.begin(), a.end(), [](auto const& p, auto const& q){
    if(p.a != q.a) return p.a < q.a;
    return p.b < q.b;
});
a.erase(unique(a.begin(), a.end(), [](auto const& p, auto const& q){
    return p.a==q.a && p.b==q.b;
}), a.end());
//作为图的边
struct Edge { int to; int w; };
vector<vector<Edge>> g(n);
g[u].push_back({v, w}); // aggregate init

//模拟(在acm打acm)
struct Team
{
```

```
    string name;
    int solved = 0;
    LL penalty = 0;
    array<bool, 13> done;
    array<int, 13> wrong;
    Team() { done.fill(false); wrong.fill(0);}
    Team(const string& s):name(s) {done.fill(false);wrong.fill(0);}
};
void Solve()
{
    int n;
    cin >> n;
    unordered_map<string, int> id;
    vector<Team> teams;
    teams.reserve(10000);
    req(i,0,n-1)
    {
        string s;
        char ch;
        int flag, t;
        cin >> s >> ch >> flag >> t;
        int pid = ch - 'A';
        if(!id.count(s))
        {
            id[s] = teams.size();
            teams.emplace_back(s);
        }
        Team &team = teams[id[s]];
        if(team.done[pid]) continue;
        if(flag == 0)
            team.wrong[pid] += 1;
        else{
            team.done[pid] = true;
            team.solved += 1;
            team.penalty += t + 20LL * team.wrong[pid];
        }
    }
    int best = 0;
    req(i,1,teams.size()-1)
    {
        if(teams[i].solved > teams[best].solved)
            best = i;
        else if(teams[i].solved == teams[best].solved && teams[i].penalty < teams[best].penalty)
```

```
            best = i;
    }
    cout << teams[best].name << " " << teams[best].solved << " " << teams[best].penalty << endl;
}
//乒乓球记分
void work(int lim)
{
    for(char p : s)
    {
        if(p == 'W') a++;
        else if(p == 'L') b++;
        if(max(a,b)>=lim && abs(a-b)>=2){
            cout << a << ":" << b << endl; a = 0; b = 0;
        }
    }
    cout << a << ":" << b << endl; a = 0; b = 0;
    cout << endl;
}
void Solve()
{

    while(cin >> m)
    {
        if (m == 'E')
            break;
        s += m;
    }
    work(11), work(21);
}
```

打表

打表的标准流程（步骤化）

1. 明确变量和范围：先选一个能穷尽小规模行为的范围（比如 n≤20、参数 m≤50）。
2. 写暴力/穷举解或动态规划作为“金标准”(brute)。保证小规模下正确性。
3. 生成表格：至少包含输入 n、输出 ans，以及可能的中间量（分解方式、余数、二进制、因子、差分、gcd 等）。
4. 观察并尝试多种视角：差分序列、比值、因数分解、二进制位、模 p 余数、最优解结构（构造法）等。

5. 基于表格猜想规律（形态：线性/多项式/指数/周期/分块/贪心），尝试证明（交换论证/极值法/归纳/不等式/反证）。
6. 用对拍扩大测试：把猜想的 O(1) 算法与暴力在更大随机测试上对比，找反例并回到第 3 步修正猜想。

表格该怎么看（常用“变换”）

1. 差分：di = ai - ai-1，反复做差分可判定多项式程度（若 kk 次差分为常数，则原序列为 kk 次多项式）。
2. 比值：ai/ai-1，看是否趋近常数（指数型）。
3. 余数/周期：看 $a_i \bmod m$ 是否周期性。
4. 二进制/lowbit：检查 ai 的二进制模式（是否与位运算相关）。
5. 因式分解 / 素因子分布：观察是否总有大因子或多次出现某素数。
6. 贪心构造跟交换法：若猜到最优由若干基本块构成，尝试用交换证明“把两个块换成其他组合是否更优”。

```
//先写出暴力做法在n<=3e4能跑出来 O(n^2)
#!/usr/bin/env python3
import random, sys
ri = lambda a,b: random.randint(a,b)  # 随机整数
ra = lambda n,mn,mx: [ri(mn,mx) for _ in range(n)]  # 随机数组
rp = lambda n: random.sample(range(1,n+1), n)  # 随机排列
rs = lambda n: ''.join(chr(ri(97,122)) for _ in range(n))  # 随机小写串
rb = lambda n: ''.join('01'[ri(0,1)] for _ in range(n))  # 随机二进制串
def rt(n):  # 随机树
    e = []
    for i in range(2, n+1):
        e.append((ri(1,i-1), i))
    random.shuffle(e)
    return e
def rg(n,m,d=0):  # 随机图
    e = set()
    while len(e) < m:
        u,v = ri(1,n),ri(1,n)
        if u==v: continue
        if not d and u>v: u,v=v,u
        e.add((u,v))
    return list(e)
# ====== 快速生成测试数据 ======
if __name__ == "__main__":
    random.seed(123)  # 固定种子

    # 1. 数组 + 查询
    # n, q = 100000, 100000
    # print(n, q)
    # print(*ra(n, 1, 10**9))
    # for _ in range(q):
    #     print(ri(1,n), ri(1,n))

    # 2. 树
    # n = 100000
    # print(n)
    # for u,v in rt(n):
    #     print(u, v)

    # 3. 图
    # n, m = 1000, 2000
    # print(n, m)
    # for u,v in rg(n, m):
```

```
    #     print(u, v)

    # 4. 字符串
    # n = 100000
    # print(rs(n))

    # 5. 排列
    # n = 100000
    # print(n)
    # print(*rp(n))
```

时间复杂度速查

```
n ≤ 10: O(n!) 全排列
n ≤ 20: O(2^n) 状态压缩
n ≤ 100: O(n^3) Floyd
n ≤ 1000: O(n^2) DP
n ≤ 10^5: O(nlogn) 排序、线段树
n ≤ 10^6: O(n) 双指针、单调栈
```

常见错误

1. **整数溢出**：中间结果可能溢出，及时取模或开long long
2. **浮点数精度**：避免直接比较，使用eps
3. **递归爆栈**：DFS深度大时要改非递归或设置栈大小
4. **多组数据**：清空全局变量和容器
5. **下标从0/1开始**：统一标准