ACM基本算法及数据结构模版

author: Xiangrui Yu in 2022.7

目录：

1.基础算法

1.1 排序

1.1.1快速排序

```
void quick_sort(int a[],int l,int r) {
    if(l>=r) return;
    int x=a[l+r>>1],i=l-1,j=r+1;
    while(i<j) {
        do i++; while(a[i]<x);
        do j--; while(a[j]>x);
        if(i<j) swap(a[i],a[j]);
    }
    quick_sort(a,l,j);
    quick_sort(a,j+1,r);
}
```

1.1.2归并排序

```
void merge_sort(int a[],int l,int r) {
    if(l>=r) return;

    int mid=l+r>>1;
    merge_sort(a,l,mid);
    merge_sort(a,mid+1,r);

    int k=0,i=l,j=mid+1;
    while(i<=mid&&j<=r) {
        if(a[i]<=a[j]) tmp[k++]=a[i++];
        else tmp[k++]=a[j++];
    }
    while(i<=mid) tmp[k++]=a[i++];
    while(j<=r) tmp[k++]=a[j++];
    for(int i=l,j=0;i<=r;i++,j++) a[i]=tmp[j];
}
```

1.1.3 STL::sort

```
int a[N];
bool cmp(int x,int y) {//重载比较函数
    return x>y;
}
sort(a,a+n,cmp);//给下标为0~n-1的数从大到小排序
```

1.2 二分

二分模板一共有两个，分别适用于不同情况。 算法思路：假设目标值在闭区间`[l, r]`中， 每次将区间长度缩小一半，当$l = r$时，我们就找到了目标值。

1.2.1版本1

当我们将区间`[l, r]`划分成`[l, mid]`和`[mid + 1, r]`时，其更新操作是$r = mid$或者$l = mid + 1$，计算mid时不需要加1。

C++ 代码模板：

```
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}
```

1.2.2版本2

当我们将区间`[l, r]`划分成`[l, mid - 1]`和`[mid, r]`时，其更新操作是$r = mid - 1$或者$l = mid$，此时为了防止死循环，计算mid时需要加1。

C++ 代码模板：

```
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

1.2.3例题

给定一个按照升序排列的长度为n的整数数组，以及q个查询。

对于每个查询，返回一个元素k的起始位置和终止位置（位置从 0 开始计数）。

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

using namespace std;

const int N=100010;

int a[N],n,k;

int main() {
    cin>>n>>k;
    for(int i=1;i<=n;i++) cin>>a[i];
    while(k--) {
        int x;cin>>x;
        int l=1,r=n;
        while(l<r) {
            int mid=l+r>>1;
            if(a[mid]>=x) r=mid;
            else l=mid+1;
        }
        if(a[l]!=x) {
            cout<<-1<<" "<<-1<<endl;continue;
        }//找到第一个大于等于x的位置
        cout<<l-1<<" ";
        l=1,r=n;
        while(l<r) {
            int mid=l+r+1>>1;
            if(a[mid]<=x) l=mid;
            else r=mid-1;
        }//找到第一个小于等于x的位置
        cout<<l-1<<endl;
    }
}
```

1.3 高精度

1.3.1高精度加法

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

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
        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;
}

int main()
{
    string a, b;
    vector<int> A, B;
    cin >> a >> b;
    for (int i = a.size() - 1; i >= 0; i -- ) A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; i -- ) B.push_back(b[i] - '0');

    auto C = add(A, B);

    for (int i = C.size() - 1; i >= 0; i -- ) cout << C[i];
    cout << endl;

    return 0;
}
```

1.3.2高精度减法

```
bool cmp(vector<int> &A, vector<int> &B)
{
    if (A.size() != B.size()) return A.size() > B.size();

    for (int i = A.size() - 1; i >= 0; i -- )
        if (A[i] != B[i])
            return A[i] > B[i];

    return true;
```

```
}

vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i ++ )
    {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}

int main()
{
    string a, b;
    vector<int> A, B;
    cin >> a >> b;
    for (int i = a.size() - 1; i >= 0; i -- ) A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; i -- ) B.push_back(b[i] - '0');

    vector<int> C;

    if (cmp(A, B)) C = sub(A, B);
    else C = sub(B, A), cout << '-';

    for (int i = C.size() - 1; i >= 0; i -- ) cout << C[i];
    cout << endl;

    return 0;
}
```

1.3.3高精度乘法

```
vector<int> mul(vector<int> &A, int b)
{
    vector<int> C;

    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ )
    {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }
```

```
    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}

int main()
{
    string a;
    int b;

    cin >> a >> b;

    vector<int> A;
    for (int i = a.size() - 1; i >= 0; i -- ) A.push_back(a[i] - '0');

    auto C = mul(A, b);

    for (int i = C.size() - 1; i >= 0; i -- ) printf("%d", C[i]);

    return 0;
}
```

1.3.4高精度除法

```
vector<int> div(vector<int> &A, int b, int &r)
{
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- )
    {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}

int main()
{
    string a;
    vector<int> A;

    int B;
    cin >> a >> B;
    for (int i = a.size() - 1; i >= 0; i -- ) A.push_back(a[i] - '0');

    int r;
    auto C = div(A, B, r);
```

```
    for (int i = C.size() - 1; i >= 0; i -- ) cout << C[i];

    cout << endl << r << endl;

    return 0;
}
```

1.4 RMQ

查询区间最小/最大

```
const int N = 200010, M = 18;

int n, m;
int w[N];
int f[N][M];

void init()
{
    for (int j = 0; j < M; j ++ )
        for (int i = 1; i + (1 << j) - 1 <= n; i ++ )
            if (!j) f[i][j] = w[i];
            else f[i][j] = max(f[i][j - 1], f[i + (1 << j - 1)][j - 1]);
}

int query(int l, int r)
{
    int len = r - l + 1;
    int k = log(len) / log(2);

    return max(f[l][k], f[r - (1 << k) + 1][k]);
}

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &w[i]);

    init();

    scanf("%d", &m);
    while (m -- )
    {
        int l, r;
        scanf("%d%d", &l, &r);
        printf("%d\n", query(l, r));
    }

    return 0;
}
```

2.数据结构

2.1 栈

2.1.1 Stack

```
const int N = 100010;
int m;
int stk[N], tt;
int main()
{
    cin >> m;
    while (m -- )
    {
        string op;
        int x;

        cin >> op;
        if (op == "push")
        {
            cin >> x;
            stk[ ++ tt] = x;
        }
        else if (op == "pop") tt -- ;
        else if (op == "empty") cout << (tt ? "NO" : "YES") << endl;
        else cout << stk[tt] << endl;
    }

    return 0;
}
```

2.1.2 单调栈

```
int n;
cin >> n;
while (n -- )
{
    int x;
    scanf("%d", &x);
    while (tt && stk[tt] >= x) tt -- ;
    if (!tt) printf("-1 ");
    else printf("%d ", stk[tt]);
    stk[ ++ tt] = x;
}
```

2.1.3 STL::Stack

2.2 堆

例题

维护一个集合，初始时集合为空，支持如下几种操作：

- `I x`，插入一个数 x；
- `PM`，输出当前集合中的最小值；
- `DM`，删除当前集合中的最小值（数据保证此时的最小值唯一）；
- `D k`，删除第 k 个插入的数；
- `C k x`，修改第 k 个插入的数，将其变为 x； 现在要进行 N 次操作，对于所有第 2 个操作，输出当前集合的最小值。

维护一个小根堆，最主要的两个操作：up(),down()

```
const int N = 100010;
int h[N], ph[N], hp[N], cnt;

void heap_swap(int a, int b)
{
    swap(ph[hp[a]],ph[hp[b]]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u)
{
    int t = u;
    if (u * 2 <= cnt && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)
    {
        heap_swap(u, t);
        down(t);
    }
}

void up(int u)
{
    while (u / 2 && h[u] < h[u / 2])
    {
        heap_swap(u, u / 2);
        u >>= 1;
    }
}

int main()
{
    int n, m = 0;
    scanf("%d", &n);
    while (n -- )
```

```
    {
        char op[5];
        int k, x;
        scanf("%s", op);
        if (!strcmp(op, "I"))
        {
            scanf("%d", &x);
            cnt ++ ;
            m ++ ;
            ph[m] = cnt, hp[cnt] = m;
            h[cnt] = x;
            up(cnt);
        }
        else if (!strcmp(op, "PM")) printf("%d\n", h[1]);
        else if (!strcmp(op, "DM"))
        {
            heap_swap(1, cnt);
            cnt -- ;
            down(1);
        }
        else if (!strcmp(op, "D"))
        {
            scanf("%d", &k);
            k = ph[k];
            heap_swap(k, cnt);
            cnt -- ;
            up(k);
            down(k);
        }
        else
        {
            scanf("%d%d", &k, &x);
            k = ph[k];
            h[k] = x;
            up(k);
            down(k);
        }
    }

    return 0;
}
```

2.3 队列

2.3.1 数组模拟队列

```
int m;
int q[N], hh, tt = -1;

int main()
{
    cin >> m;

    while (m -- )
    {
        string op;
        int x;

        cin >> op;
        if (op == "push")
        {
            cin >> x;
            q[ ++ tt] = x;
        }
        else if (op == "pop") hh ++ ;
        else if (op == "empty") cout << (hh <= tt ? "NO" : "YES") << endl;
        else cout << q[hh] << endl;
    }

    return 0;
}
```

2.3.2 单调队列

滑动区间的最大、最小值 队列中的数一定是单调递增、递减

```
const int N = 1000010;
int a[N], q[N];

int main()
{
    int n, k;
    scanf("%d%d", &n, &k);
    for (int i = 0; i < n; i ++ ) scanf("%d", &a[i]);

    int hh = 0, tt = -1;
    for (int i = 0; i < n; i ++ )
    {
        if (hh <= tt && i - k + 1 > q[hh]) hh ++ ;

        while (hh <= tt && a[q[tt]] >= a[i]) tt -- ;
        q[ ++ tt] = i;
```

```
        if (i >= k - 1) printf("%d ", a[q[hh]]);
    }

    puts("");

    hh = 0, tt = -1;
    for (int i = 0; i < n; i ++ )
    {
        if (hh <= tt && i - k + 1 > q[hh]) hh ++ ;

        while (hh <= tt && a[q[tt]] <= a[i]) tt -- ;
        q[ ++ tt] = i;

        if (i >= k - 1) printf("%d ", a[q[hh]]);
    }

    puts("");

    return 0;
}
```

STL::deque版

```
typedef pair<int,int> PII;
#define x first
#define y second
deque<PII> q;
const int N=1000010;
int a[N];
int main() {
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>a[i];

    for(int i=1;i<=n;i++) {
        while(!q.empty()&&q.front().y+m-1<i) q.pop_front();
        while(!q.empty()&&q.back().x>=a[i]) q.pop_back();
        q.push_back({a[i],i});
        if(i-m+1>=1) {
            cout<<q.front().x<<" ";
        }
    }
    cout<<endl;
    while(!q.empty()) q.pop_front();

    for(int i=1;i<=n;i++) {
        while(!q.empty()&&q.front().y+m-1<i) q.pop_front();
        while(!q.empty()&&q.back().x<=a[i]) q.pop_back();
        q.push_back({a[i],i});
        if(i-m+1>=1) {
            cout<<q.front().x<<" ";
```

```
            }
        }
    }
```

2.4 KMP

求出模板串 P 在模式串 S 中所有出现的位置的起始下标。

```
const int N = 100010, M = 1000010;
int n, m;
int ne[N];
char s[M], p[N];

int main()
{
    cin >> n >> p + 1 >> m >> s + 1;

    for (int i = 2, j = 0; i <= n; i ++ )
    {
        while (j && p[i] != p[j + 1]) j = ne[j];
        if (p[i] == p[j + 1]) j ++ ;
        ne[i] = j;
    }

    for (int i = 1, j = 0; i <= m; i ++ )
    {
        while (j && s[i] != p[j + 1]) j = ne[j];
        if (s[i] == p[j + 1]) j ++ ;
        if (j == n)
        {
            printf("%d ", i - n);
            j = ne[j];
        }
    }
    return 0;
}
```

2.5 Trie

给n个字符串，插入和询问两种操作

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

const int N=100010;
int son[N][26],cnt[N],idx=0;

void insert(string s) {
    int p=0;
    for(auto c:s) {
        int x=c-'a';
        if(son[p][x]==0) son[p][x]=++idx;
        p=son[p][x];
    }
    cnt[p]++;
}
int query(string s) {
    int p=0;
    for(auto c:s) {
        int x=c-'a';
        if(son[p][x]==0) return 0;
        p=son[p][x];
    }
    return cnt[p];
}

int main() {
    int n;cin>>n;
    while(n--) {
        string op,str;
        cin>>op>>str;
        if(op=="I") insert(str);
        else cout<<query(str)<<endl;
    }
}
```

2.6 并查集

2.6.1 朴素版并查集

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

int main() {
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++) p[i]=i;
    for(int i=1;i<=m;i++) {
        string op;
        int x,y;
        cin>>op>>x>>y;
        int fx=find(x),fy=find(y);
        if(op=="M") p[fx]=fy;
        else {
            if(fx==fy) cout<<"Yes\n";
            else cout<<"No\n";
        }
    }
}
```

2.6.2 记录信息的并查集

并查集可以用于维护具有传递性关系的作用，每个集合的大小，绑定到根结点上，每个点到根结点的距离，绑定到每个元素的结点上

```
int find(int x)
{
    if (p[x] != x)
    {
        int root = find(p[x]);
        d[x] += d[p[x]];
        p[x] = root;
    }
    return p[x];
}
```

2.7 哈希表

2.7.1 开放寻址法

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

const int N=200003, null=0x3f3f3f3f;//一般大小设置为大于2倍的质数

int h[N];

int find(int x) {//找到第一个等于 null or x 的下标
    int t=(x%N+N)%N;
    while(h[t]!=null&&h[t]!=x) {
        t++;
        if(t==N) t=0;
    }
    return t;
}

int main() {
    memset(h,0x3f,sizeof h);
    int n;
    cin>>n;
    for(int i=1;i<=n;i++) {
        string op;
        cin>>op;
        int x;
        cin>>x;
        int t=find(x);
        if(op=="I") h[t]=x;
        else{
            if(h[t]==null) cout<<"No\n";
            else cout<<"Yes\n";
        }
    }
}
```

2.7.2 拉链法

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

const int N=100003;//大于size的第一个质数
int h[N],e[N],ne[N],idx;

void insert(int x) {
    int t=(x%N+N)%N;
    e[idx]=x;
```

```
    ne[idx]=h[t];
    h[t]=idx++;
}

int find(int x) {
    int t=(x%N+N)%N;
    for(int i=h[t];~i;i=ne[i]){
        if(e[i]==x) return 1;
    }
    return 0;
}

int main() {
    memset(h,-1,sizeof h);
    int n;
    cin>>n;
    for(int i=1;i<=n;i++) {
        string op;
        cin>>op;
        int x;
        cin>>x;
        if(op=="I") insert(x);
        else {
            if(find(x)) cout<<"Yes\n";
            else cout<<"No\n";
        }

    }
}
```

2.7.3 字符串哈希

```
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

2.8 线段树

2.8.1 区间最大值

```
const int N = 200010;

int m, p;
struct Node
{
    int l, r;
    int v;  // 区间[l, r]中的最大值
}tr[N * 4];

void pushup(int u)  // 由子节点的信息，来计算父节点的信息
{
    tr[u].v = max(tr[u << 1].v, tr[u << 1 | 1].v);
}

void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r) return;
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}

int query(int u, int l, int r)
{
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].v;   // 树中节点，已经被完全包含在[l, r]中了

    int mid = tr[u].l + tr[u].r >> 1;
    int v = 0;
    if (l <= mid) v = query(u << 1, l, r);
    if (r > mid) v = max(v, query(u << 1 | 1, l, r));

    return v;
}

void modify(int u, int x, int v)
{
    if (tr[u].l == x && tr[u].r == x) tr[u].v = v;
    else
    {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) modify(u << 1, x, v);
        else modify(u << 1 | 1, x, v);
        pushup(u);
    }
}
```

```
int main()
{
    int n = 0, last = 0;
    scanf("%d%d", &m, &p);
    build(1, 1, m);

    int x;
    char op[2];
    while (m -- )
    {
        scanf("%s%d", op, &x);
        if (*op == 'Q')
        {
            last = query(1, n - x + 1, n);
            printf("%d\n", last);
        }
        else
        {
            modify(1, n + 1, ((LL)last + x) % p);
            n ++ ;
        }
    }

    return 0;
}
```

2.8.2 维护区间和

老师交给小可可一个维护数列的任务，现在小可可希望你来帮他完成。

有长为 N 的数列，不妨设为 a1,a2,...,aN。

有如下三种操作形式：

1. 把数列中的一段数全部乘一个值；
2. 把数列中的一段数全部加一个值；
3. 询问数列中的一段数的和，由于答案可能很大，你只需输出这个数模 P 的值。

- 操作 1：`1 t g c`，表示把所有满足 t≤i≤g 的 ai 改为 ai×c；
- 操作 2：`2 t g c`，表示把所有满足 t≤i≤g 的 ai 改为 ai+c；
- 操作 3：`3 t g`，询问所有满足 t≤i≤g 的 ai 的和模 P 的值。

```
typedef long long LL;

const int N = 100010;

int n, p, m;
int w[N];
struct Node
{
```

```
    int l, r;
    int sum, add, mul;
}tr[N * 4];

void pushup(int u)
{
    tr[u].sum = (tr[u << 1].sum + tr[u << 1 | 1].sum) % p;
}

void eval(Node &t, int add, int mul)
{
    t.sum = ((LL)t.sum * mul + (LL)(t.r - t.l + 1) * add) % p;
    t.mul = (LL)t.mul * mul % p;
    t.add = ((LL)t.add * mul + add) % p;
}

void pushdown(int u)
{
    eval(tr[u << 1], tr[u].add, tr[u].mul);
    eval(tr[u << 1 | 1], tr[u].add, tr[u].mul);
    tr[u].add = 0, tr[u].mul = 1;
}

void build(int u, int l, int r)
{
    if (l == r) tr[u] = {l, r, w[r], 0, 1};
    else
    {
        tr[u] = {l, r, 0, 0, 1};
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}

void modify(int u, int l, int r, int add, int mul)
{
    if (tr[u].l >= l && tr[u].r <= r) eval(tr[u], add, mul);
    else
    {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, add, mul);
        if (r > mid) modify(u << 1 | 1, l, r, add, mul);
        pushup(u);
    }
}

int query(int u, int l, int r)
{
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;

    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
```

```
    int sum = 0;
    if (l <= mid) sum = query(u << 1, l, r);
    if (r > mid) sum = (sum + query(u << 1 | 1, l, r)) % p;
    return sum;
}

int main()
{
    scanf("%d%d", &n, &p);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &w[i]);
    build(1, 1, n);

    scanf("%d", &m);
    while (m -- )
    {
        int t, l, r, d;
        scanf("%d%d%d", &t, &l, &r);
        if (t == 1)
        {
            scanf("%d", &d);
            modify(1, l, r, 0, d);
        }
        else if (t == 2)
        {
            scanf("%d", &d);
            modify(1, l, r, d, 1);
        }
        else printf("%d\n", query(1, l, r));
    }

    return 0;
}
```

3.图论

3.1 最短路

3.1.1 floyd多源最短路

```
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]=min(dis[i][k]+dis[k][j],dis[i][j]);
            }
        }
    }
}
```

3.1.2 Dijkstra单源最短路

```
#include<bits/stdc++.h>
using namespace std;
typedef pair<int,int> PII;
const int N=150010;
int h[N],e[N],ne[N],f[N],idx;
void add(int x,int y,int z) {
    e[idx]=y,f[idx]=z,ne[idx]=h[x],h[x]=idx++;
}
int n,m;
int dis[N],st[N];

int dijkstra() {
    memset(dis,0x3f,sizeof dis);
    priority_queue<PII,vector<PII>,greater<PII>> q;
    q.push({0,1});
    dis[1]=0;
    while(!q.empty()) {
        auto cur=q.top();
        q.pop();
        int distance=cur.first,x=cur.second;
        if(st[x]) continue;
        st[x]=1;
        for(int i=h[x];~i;i=ne[i]) {
            int y=e[i],z=f[i];
            if(dis[y]>distance+z) {
                dis[y]=distance+z;
                q.push({dis[y],y});
            }
        }
    }
```

```
    return dis[n];
}

int main() {
    memset(h,-1,sizeof h);
    cin>>n>>m;
    for(int i=0;i<m;i++) {
        int x,y,z;
        cin>>x>>y>>z;
        add(x,y,z);
    }
    int res=dijkstra();
    if(res==0x3f3f3f3f) cout<<-1;
    else cout<<res;
}
```

3.1.3 bellman-ford单源最短路

```
void bellman_ford() {
    memset(dis,0x3f,sizeof dis);
    dis[1]=0;
    for(int i=1;i<=k;i++) {
        memcpy(last,dis,sizeof dis);
        for(int j=1;j<=m;j++) {
            int x=e[j].x,y=e[j].y,z=e[j].z;
            dis[y]=min(dis[y],last[x]+z);
        }
    }
    if(dis[n] > 0x3f3f3f3f/2) cout<<"impossible";
    else cout<<dis[n];
}
```

3.1.4 spfa单源最短路

spfa也能判负环

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。
```

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

    while (q.size())
    {
        auto 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;       // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

3.2 最小生成树

3.2.1 kruskal算法

```
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)     // 如果两个连通块不连通，则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

3.2.2 prim算法

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

3.3 二分图

3.3.1 染色法判二分图

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

3.3.2 计算二分图最大匹配-匈牙利算法

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

const int N=510,M=100010;
int h[N],e[M],ne[M],idx;
void add(int x,int y){
    e[idx]=y,ne[idx]=h[x],h[x]=idx++;
}
```

```
int n1,n2,m;
int st[N],match[N];

int find(int x) {
    for(int i=h[x];~i;i=ne[i]) {
        int y=e[i];
        if(st[y]==1) continue;
        st[y]=1;
        if(match[y]==0||find(match[y])==1) {
            match[y]=x;
            return 1;
        }
    }
    return 0;
}

int main() {
    memset(h,-1,sizeof h);
    cin>>n1>>n2>>m;
    while(m--) {
        int x,y;
        cin>>x>>y;
        add(x,y);
    }
    int res=0;
    for(int i=1;i<=n1;i++) {
        memset(st,0,sizeof st);
        res+=find(i);
    }
    cout<<res;
}
```

3.3.3 带权二分图的最大匹配-KM算法

```
//前提是匹配数最大
//时间复杂度O(N^4)且要求左右部点均为n个
#define maxn 105
int w[maxn][maxn];//边权
int la[maxn],lb[maxn];//左、右部点的顶标
bool va[maxn],vb[maxn];//访问标记：是否在交错树中
int match[maxn];//右部点匹配了哪一个左部点
int n,delta,upd[maxn];
bool dfs(int x) {
    va[x] = 1;
    for (int y = 1; y <= n; y++) {
        if (!vb[y]) {
            if (la[x] + lb[y] - w[x][y] == 0) {
                vb[y] = 1;
                if (!match[y] || dfs(match[y])) {
                    match[y] = x;
                    return true;
```

```
                }
            } else upd[y] = min(upd[y], la[x] + lb[y] - w[x][y]);
        }
    }
    return false;
}
int KM() {
    for(int i=1;i<=n;i++) {
    match[i]=0;
        la[i]=-inf;
        lb[i]=0;
        for(int j=1;j<=n;j++) la[i]=max(la[i],w[i][j]);
    }
    for(int i=1;i<=n;i++) {
        while (true) {
            delta=inf;
            memset(va,0,sizeof(va));
            memset(vb,0,sizeof(vb));
            for(int j=1;j<=n;j++) upd[j]=inf;
            if(dfs(i)) break;
            for(int j=1;j<=n;j++) {
                if(!vb[j]) delta=min(delta,upd[j]);
            }
            for(int j=1;j<=n;j++) {
                if(va[j]) la[j]-=delta;
                if(vb[j]) lb[j]+=delta;
            }
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++) ans+=w[match[i]][i];
    return ans;
}
```

3.4 有向图的强联通分量

3.4.1 tarjan算法

```
#define maxn 105
vector<int> v[maxn];
int dfn[maxn],low[maxn],col[maxn],instack[maxn];
stack<int> s;
int block=1;
int tot=0;
void tarjan(int x) {
    low[x]=dfn[x]=++tot;//关键
    s.push(x);instack[x]=1;
    for(int i=0;i<v[x].size();i++) {
        int to=v[x][i];
        if(!dfn[to]) {
            tarjan(to);
            low[x]=min(low[x],low[to]);
        }
        else if(instack[to])//能够到达的树上节点
            low[x]=min(low[x],dfn[to]);
    }
    if(low[x]==dfn[x]) {
        while (s.top()!=x){
            col[s.top()]=block;
            instack[s.top()]=0;
            s.pop();
        }
        col[s.top()]=block++;
        instack[s.top()]=0;
        s.pop();
    }
}
```

3.4.2 求割点

```
#define maxn 20000+5
int dfn[maxn],low[maxn],tot;
int instack[maxn];
vector<int> v[maxn];
stack<int> s;
int cut[maxn];
void tarjan(int x,int root) {
    dfn[x]=low[x]=++tot;
    s.push(x);instack[x]=1;
    int child=0;
    for(int i=0;i<v[x].size();i++) {
        int to=v[x][i];
        if(!dfn[to]) {
```

```
            if(x==root) child++;
            tarjan(to,root);
            low[x]=min(low[x],low[to]);
            if(x!=root&&low[to]>=dfn[x]) cut[x]=1;
        }
        else if(instack[to]){
            low[x]=min(low[x],dfn[to]);
        }
    }
    if(low[x]==dfn[x]) {
        while (s.top()!=x){
            instack[s.top()]=0;
            s.pop();
        }
        instack[s.top()]=0;
        s.pop();
    }
    if(x==root&&child>=2) cut[x]=1;
}
```

3.5 无向图的双连通分量

3.5.1 tarjan求桥

```
const int N = 5010, M = 20010;

int n, m;
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
int stk[N], top;
int id[N], dcc_cnt;
bool is_bridge[M];
int d[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

void tarjan(int u, int from)
{
    dfn[u] = low[u] = ++ timestamp;
    stk[ ++ top] = u;

    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j, i);
            low[u] = min(low[u], low[j]);
            if (dfn[u] < low[j])
                is_bridge[i] = is_bridge[i ^ 1] = true;
        }
        else if (i != (from ^ 1))
            low[u] = min(low[u], dfn[j]);
    }

    if (dfn[u] == low[u])
    {
        ++ dcc_cnt;
        int y;
        do {
            y = stk[top -- ];
            id[y] = dcc_cnt;
        } while (y != u);
    }
}

int main()
{
```

```
    cin >> n >> m;
    memset(h, -1, sizeof h);
    while (m -- )
    {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }

    tarjan(1, -1);

    for (int i = 0; i < idx; i ++ )
        if (is_bridge[i])
            d[id[e[i]]] ++ ;

    int cnt = 0;
    for (int i = 1; i <= dcc_cnt; i ++ )
        if (d[i] == 1)
            cnt ++ ;

    printf("%d\n", (cnt + 1) / 2);

    return 0;
}
```

3.6 欧拉回路、欧拉路径

```
void dfs(int u)
{
    for (int &i = h[u]; ~i;)
    {
        if (used[i])
        {
            i = ne[i];
            continue;
        }

        used[i] = true;

        int j = e[i];
        i = ne[i];
        dfs(j);

        ans[ ++ cnt] = t;
    }
}
```

3.7 拓扑排序

```
bool topsort()
{
    int hh = 0, tt = -1;

    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }

    // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
    return tt == n - 1;
}
```

3.8 网络流

3.8.1 Dinic 最大流

```
//#pragma GCC optimize(2)
#include "bits/stdc++.h"
using namespace std;
inline int read()
{
    int x=0,f=1;char ch=getchar();
    while (ch<'0'||ch>'9'){if (ch=='-') f=-1;ch=getchar();}
    while (ch>='0'&&ch<='9'){x=x*10+ch-48;ch=getchar();}
    return x*f;
}
#define ll long long
#define int ll
#define inf 0x3f3f3f3f
const int Ni = 200005;
const int MAX = 1<<26;
int e[Ni*2],f[Ni*2],ne[Ni*2],h[Ni*2];
int idx=0;
int d[Ni],q[Ni],cur[Ni];
int S,T;
void add(int a, int b, int c){
    e[idx] = b, f[idx] = c, ne[idx] = h[a], h[a] = idx++;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs(){
    int hh = 0, tt = -1;
    memset(d, -1, sizeof(d));
    q[++tt] = S, d[S] = 0, cur[S] = h[S];
    while(hh <= tt){
        int t = q[hh++];
        for(int i = h[t]; ~i; i = ne[i]){
            int j = e[i];
            if (d[j] == -1 && f[i]){
                d[j] = d[t] + 1;
                cur[j] = h[j];
                if (j == T) return true;
                q[++tt] = j;
            }
        }
    }
    return false;
}

int find(int u, int limit){
    if (u == T) return limit;
    int flow = 0;
    // start from cur[u] instead of h[u] <- important
```

```
    for(int i = cur[u]; ~i && flow < limit; i = ne[i]){
        int j = e[i];
        cur[u] = i;
        if (d[j] == d[u] + 1 && f[i]){
            int t = find(j, min(f[i], limit - flow));
            if (!t) d[j] = -1;
            else f[i] -= t, f[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}

int dinic(){
    int res = 0, flow;
    while(bfs()) while(flow = find(S, inf)) res += flow;
    return res;
}
signed main() {
    int n,m;
    n=read();m=read();S=read();T=read();
    idx=0;memset(h,-1,sizeof(h));
    for(int i=1;i<=m;i++) {
        int x,y,z;
        x=read();y=read();z=read();
        add(x,y,z);
    }
    cout<<dinic();
}
```

4.数论

4.1 质数

4.1.1 试除法求质数

```
bool is_prime(int x)
{
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
```

4.1.2 试除法分解质因数

```
void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
```

4.1.3 朴素筛筛质数

```
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]) continue;
        primes[cnt ++ ] = i;
        for (int j = i + i; j <= n; j += i)
            st[j] = true;
    }
}
```

4.1.4 线性筛筛质数

```
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;
        }
    }
}
```

4.2 约数

4.2.1 试除法求所有约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end());
    return res;
}
```

4.2.2 约数个数、约数之和

如果 $N = p1^{c1} * p2^{c2} * ... * pk^{ck}$ 约数个数： $(c1 + 1) * (c2 + 1) * ... * (ck + 1)$ 约数之和： $(p1^0 + p1^1 + ... + p1^{c1}) * ... * (pk^0 + pk^1 + ... + pk^{ck})$

4.3 最大公因数

4.3.1 欧几里得算法

```
int gcd(int a, int b)
{
    return b ? gcd(b, a % b) : a;
}
```

4.3.2 求欧拉函数

```
int phi(int x)
{
    int res = x;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);

    return res;
}
```

4.3.3 筛法求欧拉函数

```
int primes[N], cnt;     // primes[]存储所有素数
int euler[N];           // 存储每个数的欧拉函数
bool st[N];         // st[x]存储x是否被筛掉

void get_eulers(int n)
{
    euler[1] = 1;
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i])
        {
            primes[cnt ++ ] = i;
            euler[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            int t = primes[j] * i;
            st[t] = true;
            if (i % primes[j] == 0)
```

```
                {
                    euler[t] = euler[i] * primes[j];
                    break;
                }
                euler[t] = euler[i] * (primes[j] - 1);
            }
        }
    }
```

4.3.4 扩展欧几里得算法

```
// 求x, y，使得ax + by = gcd(a, b)
int exgcd(int a, int b, int &x, int &y)
{
    if (!b)
    {
        x = 1; y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a/b) * x;
    return d;
}
```

4.4 快速幂

```
//求 m^k mod p，时间复杂度 O(logk)。

int qmi(int m, int k, int p)
{
    int res = 1 % p, t = m;
    while (k)
    {
        if (k&1) res = res * t % p;
        t = t * t % p;
        k >>= 1;
    }
    return res;
}
```

4.5 高斯消元

```
// a[N][N]是增广矩阵
int gauss()
{
    int c, r;
    for (c = 0, r = 0; c < n; c ++ )
    {
        int t = r;
        for (int i = r; i < n; i ++ )   // 找到绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;

        if (fabs(a[t][c]) < eps) continue;

        for (int i = c; i <= n; i ++ ) swap(a[t][i], a[r][i]);      // 将绝对值最大的行换到最顶端
        for (int i = n; i >= c; i -- ) a[r][i] /= a[r][c];      // 将当前行的首位变成1
        for (int i = r + 1; i < n; i ++ )       // 用当前行将下面所有的列消成0
            if (fabs(a[i][c]) > eps)
                for (int j = n; j >= c; j -- )
                    a[i][j] -= a[r][j] * a[i][c];

        r ++ ;
    }

    if (r < n)
    {
        for (int i = r; i < n; i ++ )
            if (fabs(a[i][n]) > eps)
                return 2; // 无解
        return 1; // 有无穷多组解
    }

    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];

    return 0; // 有唯一解
}
```

4.6 组合数

4.6.1 递推法

```
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

4.6.2 处理逆元方法求组合数

```
首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]
如果取模的数是质数，可以用费马小定理求逆元
int qmi(int a, int k, int p)    // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

// 预处理阶乘的余数和阶乘逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i ++ )
{
    fact[i] = (LL)fact[i - 1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}
```

4.6.3 Lucas定理

```
若p是质数，则对于任意整数 1 <= m <= n，有：
    C(n, m) = C(n % p, m % p) * C(n / p, m / p) (mod p)

int qmi(int a, int k, int p)  // 快速幂模板
{
    int res = 1 % p;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
```

```
        k >>= 1;
    }
    return res;
}

int C(int a, int b, int p)  // 通过定理求组合数C(a, b)
{
    if (a < b) return 0;

    LL x = 1, y = 1;  // x是分子，y是分母
    for (int i = a, j = 1; j <= b; i --, j ++ )
    {
        x = (LL)x * i % p;
        y = (LL) y * j % p;
    }

    return x * (LL)qmi(y, p - 2, p) % p;
}

int lucas(LL a, LL b, int p)
{
    if (a < p && b < p) return C(a, b, p);
    return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
}
```

4.6.4 分解质因数法求组合数

```
当我们需要求出组合数的真实值，而非对某个数的余数时，分解质因数的方式比较好用：
    1. 筛法求出范围内的所有质数
    2. 通过 C(a, b) = a! / b! / (a - b)! 这个公式求出每个质因子的次数。 n! 中p的
次数是 n / p + n / p^2 + n / p^3 + ...
    3. 用高精度乘法将所有质因子相乘

int primes[N], cnt;     // 存储所有质数
int sum[N];     // 存储每个质数的次数
bool st[N];     // 存储每个数是否已被筛掉


void get_primes(int n)      // 线性筛法求素数
{
    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;
        }
    }
}
```

```
int get(int n, int p)       // 求n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}


vector<int> mul(vector<int> a, int b)       // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i ++ )
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
}

get_primes(a);  // 预处理范围内的所有质数

for (int i = 0; i < cnt; i ++ )     // 求每个质因数的次数
{
    int p = primes[i];
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

vector<int> res;
res.push_back(1);

for (int i = 0; i < cnt; i ++ )     // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j ++ )
        res = mul(res, primes[i]);
```

4.7 卡特兰数

```
给定n个0和n个1，它们按照某种顺序排成长度为2n的序列，满足任意前缀中0的个数都不少于1的个数的序列的数量为： Cat(n) = C(2n, n) / (n + 1)
```