

基础算法

类型	范围
int	$[-2^{31}, 2^{31}]$
long long	$[-2^{63}, 2^{63} - 1]$

基本单位

基本单位	详解
Bit (比特)	计算机中最小数据单位，表示二进制的一位
Byte (字节)	计算机存储的基本单位， 1 Byte = 8 Bits

快速排序

1. 确定分界点
2. **重点**：调整区间，根据分界点重新划分数据分布，保证左边数小于分界点，右边数大于分界点
 - 暴力做法：
 - 创建两个数组a, b
 - 扫描原数组，小于分界点的数放到a，大于分界点的数放到b
 - 然后再把a和b里面的数放回原数组
 - 优雅做法（双指针）：
 - 创建两个指针l, r，分别直指向数组的头部和尾部
 - l从左边开始向右扫描，碰到大于等于分界点的数就停下来
 - r从右边开始向左扫描，碰到小于等于分界点的数就停下来
 - 交换两个指针指向的数，然后各自移动1位
 - 直到l和r相遇
3. 递归处理左右两端

```

1  #include<iostream>
2  using namespace std;
3  const int N = 1e6 + 10;
4
5  int n;
6  int q[N];
7
8  void quick_sort(int q[],int l,int r){
9      if(l >= r) return;
10     int x = q[(int)((l + r) / 2)];
11     int i = l - 1 , j = r + 1;
12     while(i < j){
13         do i++; while(q[i] < x);
14         do j--; while(q[j] > x);
15         if(i < j) swap(q[i],q[j]);
16     }
17     quick_sort(q,l,j);
18     quick_sort(q,j + 1,r);
19 }
20
21 int main(){
22     scanf("%d", &n);
23     for(int i = 0;i < n;i++) scanf("%d",&q[i]);
24
25     quick_sort(q,0,n - 1);
26
27     for(int i = 0;i < n;i++) printf("%d ",q[i]);
28
29     return 0;
30 }

```

归并排序

- 时间复杂度： $O(n \log n)$
- 1. 确定分离点： $mid = \frac{l+r}{2}$
- 2. 递归排序数组左部分和右部分
- 3. 归并排序完的数组，合二为一
 - 用双指针，分别指向两个已排序好的数组的第一位
 - 比较指针指向的数，更小的数写入答案数组
 - 当访问完任意一个数组后退出循环

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 1e6 + 10;
5  int n;
6  int q[N],temp[N];
7
8  void merge_sort(int q[],int l,int r){
9      if(l >= r) return;
10     int mid = (l + r) / 2;
11     merge_sort(q, l, mid);
12     merge_sort(q, mid + 1, r);
13
14     int k = 0,i = l,j = mid + 1;
15     while(i <= mid && j <= r){
16         if(q[i] <= q[j]) temp[k++] = q[i++];
17         if(q[i] > q[j]) temp[k++] = q[j++];
18     }
19     while(i <= mid) temp[k++] = q[i++];
20     while(j <= r) temp[k++] = q[j++];
21
22     for(int i = l, j = 0;i <= r;i++,j++) q[i] = temp[j];
23 }
24
25
26 int main(){
27     scanf("%d",&n);
28
29     for(int i = 0;i < n;i++) scanf("%d",&q[i]);
30
31     merge_sort(q,0,n - 1);
32
33     for(int i = 0;i < n;i++) printf("%d ",q[i]);
34
35     return 0;
36 }

```

整数二分

```

1 // 检查查找的数是否满足性质
2 bool check(int x){...}
3
4 // 查找最小的满足check条件的值
5 int binary_search1(int l,int r){
6     while(l < r){
7         int mid = (l + r) / 2;
8         if(check(mid)) r = mid;
9         else l = mid + 1;
10    }
11    return l;
12 }
13
14
15 // 查找最大的满足check条件的值
16 int binary_search2(int l, int r){
17     while(l < r){
18         int mid = (l + r + 1) / 2; // 加1是防止死循环
19         if(check(mid)) l = mid;
20         else r = mid - 1;
21     }
22     return l;
23 }

```

浮点数二分

```

1 bool check(double x){}
2
3 // 不需要处理边界
4 double binary_search(double l, double r){
5     while(r - l > eps){ // eps表示精度，取决题目对精度要求
6         double mid = (l + r) / 2;
7         if(check(mid)) r = mid;
8         else l = mid;
9     }
10    return l;
11 }

```

高精度

也就是把大整数的每一位存到数组里，个位存到数组的第一个位置，后面以此类推

- 加法：

```

1  #include<iostream>
2  #include<vector>
3  const int N = 1e6 + 10;
4  int n;
5  using namespace std;
6
7  vector<int> add(vector<int> &a,vector<int> &b) { //加引用可以提高效率
8      vector<int> c;
9      int t = 0; //进位
10     for(int i = 0;i < a.size() || i < b.size();i++){
11         if(i < a.size()) t += a[i];
12         if(i < b.size()) t += b[i];
13         c.push_back(t % 10);
14         t /= 10;
15     }
16     if(t) c.push_back(1);
17     return c
18 }
19 int main(){
20     string a, b;
21     vector<int> A, B;
22     cin >> a >> b;
23     for(int i = a.size() - 1 ;i >= 0;i--) A.push_back(a[i] - '0');
24     for(int i = b.size() - 1 ;i >= 0;i--) A.push_back(b[i] - '0');
25     vector<int> c = add(A,B);
26     for(int i = c.size() - 1;i >= 0 ;i--) printf("%d",c[i]);
27 }

```

- 减法:

- $A > B$: 算 $A - B$
- $A < B$: 算 $-(B - A)$

```

1  #include<iostream>
2  #include<vector>
3  const int N = 1e6 + 10;
4  int n;
5  using namespace std;
6
7  // 判断A是不是大于等于B
8  bool cmp(vector<int> &a,vector<int> &b){
9      if(a.size() != b.size()) return a.size() > b.size();
10     for(int i = a.size() - 1;i >= 0;i++){
11         if(a[i] != b[i]) return a[i] > b[i];
12     }
13     return true;
14 }
15
16
17 vector<int> sub(vector<int> &a,vector<int> &b){
18     vector<int> c;
19     int t = 0;
20     for(int i = 0; i < a.size();i++){
21         t = a[i] - t;
22         if(i < b.size()) t -= b[i];
23         c.push_back((t + 10) % 10);
24         if(t < 0) t = 1;
25         else t = 0;
26     }
27     while(c.size() > 1 && c.back() == 0) c.pop_back();
28     return c;
29 }
30 int main(){
31     String a, b;
32     vector<int> A, B;
33     cin >> a >> b;
34     for(int i = a.size() - 1 ;i >= 0;i--) A.push_back(a[i] - '0');
35     for(int i = b.size() - 1 ;i >= 0;i--) B.push_back(b[i] - '0');
36     if(cmp(A,B)){
37         vector<int> c = sub(A,B);
38         for(int i = c.size() - 1;i >= 0 ;i--) printf("%d",c[i]);
39     }
40     else{
41         vector<int> c = sub(B,A);
42         printf("-");
43         for(int i = c.size() - 1;i >= 0 ;i--) printf("%d",c[i]);
44     }
45 }

```

- 乘法:

```
1 vector<int> multi(vector<int> &a,int b){
2     vector<int> c;
3     int t = 0;
4     for(int i = 0;i < a.size();i++){
5         t += a[i] * b;
6         c.push_back(t % 10);
7         t = t / 10;
8     }
9     while(t > 0){
10        c.push_back(t % 10);
11        t /= 10;
12    }
13    // if(t > 0) c.push_back(t);
14    while(c.size() > 1 && c.back() == 0) c.pop_back();
15    return c;
16 }
```

- 除法

每次push_back()的值都不会超过两位数，证明如下

$$\begin{aligned}r_k &= r_{k-1} \% b \\ r_k &\leq b - 1 \\ r_k * 10 &\leq (b - 1) * 10 \\ r_k * 10 + a &\leq 10b - 1 < 10b\end{aligned}$$

也就是说 r/b 最大是 $(10b - 1)/b$ ，最大到9

```
1 vector<int> div(vector<int> &a,int b,int &r){
2     vector<int> c;
3     r = 0;
4     for(int i = a.size() - 1;i >= 0;i--){
5         r = r * 10 + a[i];
6         c.push_back(r / b);
7         r %= b;
8     }
9     while(t > 0){
10        c.push_back(t )
11    }
12    while(c.size() > 1 && c.back() == 0) c.pop_back();
13    reverse(c.begin(),c.end());
14
15 }
```

前缀和

- 一维前缀和

给定数组 a_1, a_2, \dots, a_n

前缀和数组为 $S_i = a_1 + a_2 + a_3 + \dots + a_i$

$S_k - S_l$ 表示 $(l, k]$ 的元素总和

```
1  int t;
2  int b[n];
3  // 令s0 = 0 前缀和数组长度变为 n + 1
4  void sum(int a[]){
5      int t = 0;
6      for(int i = 0 ;i <= n;i++){
7          b[i] = t;
8          t += a[i];
9      }
10 }
```

- 二维前缀和

- 给定二维数组，二维前缀和 $S_{ij} = \sum_{m \leq i, n \leq j} a_{mn}$
- 二维数组从 $a[1][1]$ 开始存储
- $S_{ij} - S_{in} - S_{mj} + S_{mn}$ 表示在二维数组里， (i, j) 、 (m, n) 、 (i, n) 、 (m, j) 围起来的矩形的所有元素的和
- $S_{ij} = a_{ij} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1}$
- $S_{0,1}$ 、 $S_{0,0}$ 、 $S_{1,0}$ 等都是0

```
1  int s[M][N];
2  int main(){
3      s[0][0] = 0;
4      for(int i = 1;i < n;i++){
5          for(int j = 1;j < m;j++){
6              scanf("%d",&a[i][j]);
7              s[i][j] = s[i][j - 1] + a[i][j] - s[i - 1][j - 1] + s[i - 1][j];
8          }
9      }
10 }
```

差分

- 一维差分

给定 a_1, a_2, \dots, a_n , 构造 b_1, b_2, \dots, b_n , 使得 $a_i = b_1 + b_2 + \dots + b_i$, 则 b 是 a 的差分, a 是 b 的前缀和

$$b_1 = a_1, b_2 = a_2 - a_1, b_3 = a_3 - a_2, \dots, b_n = a_n - a_{n-1}$$



假设我们需要对 $[l, r]$ 区间内的 a_i 全都加上 C , 这时候如果直接扫描时间复杂度是 $O(N)$ 。但我们可以求 a 的差分 b , 让 $b_l + C$ 那么 a_l 后面的数字都会加上 C , 我们再让 $b_{r+1} - C$, 就可以让 $[l, r]$ 区间内的 a 都加上 C , 而其他地方不变, 也适用于给 $a[i]$ 赋初值

- 二维差分

原矩阵: a_{ij}

差分矩阵: b_{ij}

```

1  #include<iostream>
2  using namespace std;
3
4  int n,m,q;
5
6  const int N = 1010;
7  int a[N][N],b[N][N];
8
9  void insert(int x1,int y1,int x2,int y2,int c){
10     b[x1][y1] += c;
11     b[x2 + 1][y2 + 1] += c;
12     b[x1][y2 + 1] -= c;
13     b[x2 + 1][y1] -= c;
14 }
15
16
17 int main(){
18     cin >> n >> m >> q;
19     for(int i = 1;i <= n;i++){
20         for(int j = 1;j <= m;j++){
21             int t;
22             cin >> t;
23             insert(i,j,i,j,t);
24         }
25     }
26     while(q--){
27         int x1,y1,x2,y2,c;
28         cin >> x1 >> y1 >> x2 >> y2 >> c;
29         insert(x1,y1,x2,y2,c);
30     }
31
32     for(int i = 1;i <= n;i++){
33         for(int j = 1;j <= m;j++){
34             b[i][j] = b[i][j] + b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1]; // 差
分数组直接化为前缀和数组
35         }
36     }
37
38     for(int i = 1;i <= n;i++){
39         for(int j = 1;j <= m;j++){
40             cout << b[i][j] << " ";
41         }
42         cout << endl;
43     }
44 }

```

双指针算法

可以将 $O(n^2)$ 优化到 $O(n)$

没什么好说的，做题

位运算

n的二进制表示中第k位数字是多少

从0开始算的

1. 先把第k位移到最后一位: $n \gg k$
 2. 看个位是几: $n \& 1$
 3. $n \gg k \& 1$;
- $lowbit(x)$: 返回x的**最右边（最后）**一位1和后面的数字
 - $x = 1010$, 返回10
 - $x = 101000$, 返回1000
 - `x & -x`

离散化

将不连续的数字映射成连续的数字

值域跨度很大，但是用到的值很少，用到的值十分**稀疏**

- 要对原始数据**去重**
- 用二分找要映射的位置
- 本质上是**把数组里面的数映射成数组的下标**

区间求和

- alls数组是为了存储需要离散化的点，对其进行去重和排序是为了防止映射错误
- add和query分别存储要插入的数据和搜索的数据
- 最后进行区间搜索的时候要先**离散化**

```

1  #include<iostream>
2  #include<vector>
3  #include<algorithm>
4  using namespace std;
5
6  const int N = 3e5 + 10; // 不仅有插入的点，还有查询的点，离散化后的数组的长度要能容纳插入点和
   查询的点的长度
7  int a[N],s[N]; // 离散化后用前缀和求解
8
9  vector<int> alls; // 所有要离散化的下标
10
11
12 typedef pair<int,int> PII;
13 vector<PII> add,query;
14 int n,m;
15 // x为数轴上的点，返回的是离散化后的结果
16 int find(int x){
17     int l = 0, r = alls.size() - 1;
18     while(l < r){
19         int mid = l + r >> 1;
20         if(alls[mid] >= x) r = mid;
21         else l = mid + 1;
22     }
23     return r + 1;
24 }
25
26 int main(){
27     cin >> n >> m;
28     // 插入操作
29     for(int i = 0;i < n;i++){
30         int x, c;
31         cin >> x >> c;
32         alls.push_back(x);
33         add.push_back({x,c});
34     }
35
36     // 查找操作
37     while(m--){
38         int l,r;
39         cin >> l >> r;
40         query.push_back({l,r});
41         alls.push_back(l);
42         alls.push_back(r);
43     }
44
45     //排序、去重，不然映射出错

```

```

46     sort(alls.begin(),alls.end());
47     alls.erase(unique(alls.begin(),alls.end()),alls.end());
48
49     // 处理插入
50     for(auto x: add){
51         int t = find(x.first);
52         a[t] += x.second;
53     }
54
55     // 前缀和
56     for(int i = 1;i <= alls.size();i++){
57         s[i] = a[i] + s[i - 1];
58     }
59
60     // 输出答案
61     for(auto x: query){
62         int l = find(x.first);
63         int r = find(x.second);
64         cout << s[r] - s[l - 1] << endl;
65     }
66 }

```

区间合并

把多个有交集的区间合并成一个区间

1. 按区间左端点排序
2. 扫描区间，并把区间进行合并

```

1  #include<iostream>
2  #include<algorithm>
3  #include<vector>
4  using namespace std;
5  const int N = 100010;
6
7  vector<pair<int,int>> t;
8
9  int n;
10
11
12 void merge(vector<pair<int,int>> &s){
13     vector<pair<int,int>> temp;
14     sort(s.begin(),s.end());
15     // 用第一对数据作为初始维护区间
16     int l = s[0].first,r = s[0].second;
17     for(int i = 1;i < s.size();i++){
18         if(s[i].first > r) {
19             temp.push_back({l,r});
20             l = s[i].first;
21             r = s[i].second;
22         }
23         else r = max(r,s[i].second);// 将两种情况合二为一
24     }
25     // 循环结束后一定会有一个区间没有被处理，在这里进行处理
26     temp.push_back({l,r});
27     t = temp;
28 }
29
30
31 int main(){
32     cin >> n;
33     while(n--){
34         int a,b;
35         cin >> a >> b;
36         t.push_back({a,b});
37     }
38     merge(t);
39     cout << t.size();
40 }

```

数据结构

以数组模拟为主

链表

1. 单链表：邻接表（存储树和图）
 - $e[N]$ ：存的是节点的值
 - $ne[N]$ ：存的是节点的next指针
2. 双链表：每个节点有两个指针
 - $l[N]$ ：记录节点的左指针
 - $r[N]$ ：记录节点的右指针
3. 邻接表：n个单链表

单链表

```

1  #include<iostream>
2
3  using namespace std;
4
5  const int N = 1000010;
6  /*
7  e[N]存的是当前节点的值
8  ne[N]存的是当前节点的指针
9  head存的是头节点
10 idx存的是链表已有节点的个数
11 */
12 int e[N],ne[N];
13 int head,idx;
14
15
16 void init(){
17     head = -1;
18     idx = 0;
19 }
20
21 void add_to_head(int c){
22     e[idx] = c;
23     ne[idx] = head;
24     head = idx++;
25 }
26
27 void erase(int k){
28     // 处理k为0的情况
29     if(k == -1) {
30         head = ne[head];
31     }
32     ne[k] = ne[ne[k]];
33 }
34
35 void add(int x,int c){
36     e[idx] = c;
37     ne[idx] = ne[x];
38     ne[x] = idx++;
39 }
40
41 int main(){
42     init();
43     int m;
44     cin >> m;
45     while(m--){
46         int k,x;

```

```
47     char op;
48     cin >> op;
49     if(op == 'H'){
50         cin >> x;
51         add_to_head(x);
52     }
53     else if(op == 'D'){
54         cin >> k ;
55         erase(k - 1);
56     }
57     else if(op == 'I'){
58         cin >> k >> x;
59         add(k - 1,x);
60     }
61 }
62 for(int i = head;i != -1;i = ne[i]){
63     cout << e[i] << " ";
64 }
65
66 }
67
68
69
70
```

双链表

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int l[N],r[N],e[N];
6  int idx;
7  // 令双链表的头节点下标为0, 尾节点下标为1
8  void init(){
9      r[0] = 1,l[1] = 0;
10     idx = 2;
11 }
12 // 模拟的是在k的右侧插入
13 void add(int k,int x){
14     e[idx] = x;
15     r[idx] = r[k];
16     l[idx] = k;
17     l[r[k]] = idx;
18     r[k] = idx;
19     idx++;
20 }
21
22 void remove(int k){
23     l[r[k]] = l[k];
24     r[l[k]] = r[k];
25 }
26
27
28 int main(){
29     int m;
30     init();
31     cin >> m;
32     while(m--){
33         string op;
34         cin >> op;
35         int k,x;
36         if(op == "L"){
37             cin >> x;
38             add(0,x);
39         }
40         else if(op == "R"){
41             cin >> x;
42             add(l[1],x);
43         }
44         else if(op == "D"){
45             cin >> k;
46             remove(k + 1); // 第1个插入的数实际上idx是2

```

```
47     }
48     else if(op == "IL"){
49         cin >> k >> x;
50         add(l[k + 1],x);
51     }else if(op == "IR"){
52         cin >> k >> x;
53         add(k + 1,x);
54     }
55 }
56 for(int i = r[0];i != 1;i = r[i]){
57     cout << e[i] << " ";
58 }
59 }
```

栈

先进后出

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int s[N];
6  int top;
7
8  void init(){
9      top = -1;
10 }
11
12 bool isempty(){
13     return top == -1;
14 }
15
16 void push(int x){
17     s[++top] = x;
18 }
19
20 void pop(){
21     top--;
22 }
23
24 int query(){
25     return s[top];
26 }
27
28 int main(){
29     init();
30     int m;
31     cin >> m;
32     while(m--){
33         int x;
34         string op;
35         cin >> op;
36         if(op == "push"){
37             cin >> x;
38             push(x);
39         }
40         else if(op == "pop") pop();
41         else if(op == "empty"){
42             if(isempty()) cout << "YES" << endl;
43             else cout << "NO" << endl;
44         }
45         else if(op == "query") cout << query()<< endl;
46     }

```

47

48

}

队列

先进先出

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int s[N],top,tail;
6
7  void init(){
8      top = 0;
9      tail = -1;
10 }
11
12 void push(int x){
13     s[++tail] = x;
14 }
15
16 bool isempty(){
17     return tail < top;
18 }
19
20 void pop(){
21     top++;
22 }
23
24 int query(){
25     return s[top];
26 }
27
28
29 int main(){
30     int m;
31     init();
32     cin >> m;
33     while(m--){
34         int x;
35         string op;
36         cin >> op;
37         if(op == "push"){
38             cin >> x;
39             push(x);
40         }
41         else if(op == "pop"){
42             pop();
43         }
44         else if(op == "empty"){
45             if(isempty()) cout << "YES" << endl;
46             else cout << "NO" << endl;

```

```

47     }
48     else if( op == "query"){
49         cout << query() << endl;
50     }
51 }
52 }

```

单调栈和单调队列

单调栈：给定一个序列，求每一个数左边离他最近的数且比它小在什么地方

题解出处：<https://www.acwing.com/video/5400/>

```

1  #include<iostream>
2
3  using namespace std;
4  const int N = 1e5 + 10;
5  int st[N];
6  int top = -1;
7  int n;
8  // 先用朴素算法模拟，然后找出规律，可以用单调栈求解
9  int main(){
10     cin >> n;
11     for(int i = 0;i < n;i++){
12         int x;
13         cin >> x;
14         while(top >= 0 && st[top] >= x) top--;
15         if(top != -1) cout << st[top] << " ";
16         else cout << -1 << " ";
17         st[++top] = x;
18     }
19 }

```

单调队列：求滑动窗口最大值或者最小值

- 用一个单调（双端）队列维护，每一时刻这个单调队列都是有序的，每移动一次输出一次队头元素，本质上和单调栈思想一致，都是把不可能的答案给去掉

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 1e6 + 10;
5  int a[N],st[N];
6  int head = 0,tail = -1;
7  int main(){
8
9      int n,k;
10     cin >> n >> k;
11     for(int i = 0;i < n;i++){
12         cin >> a[i];
13     }
14     // 求最小值
15     for(int i = 0;i < n;i++){
16         if( i - k + 1 > st[head]) head ++;
17         while(tail >= head && a[st[tail]] >= a[i]) tail--;
18         st[++tail] = i;
19         if(i >= k - 1) cout << a[st[head]] << " ";
20     }
21     cout << endl;
22     // 求最大值
23     head = 0,tail = -1;
24     for(int i = 0;i < n;i++){
25         if(i - k + 1 > st[head]) head ++;
26         while(tail >= head && a[st[tail]] <= a[i]) tail--;
27         st[++tail] = i;
28         if(i >= k - 1) cout << a[st[head]] << " ";
29     }
30     cout << endl;
31 }

```

队列存值的写法

```

1  #include<iostream>
2
3  using namespace std;
4  const int N = 1e6 + 10;
5  int n,k;
6  int tail = -1,head = 0;
7  int q[N],a[N];
8
9  int main(){
10     cin >> n >> k;
11     for(int i = 1;i <= n;i++){
12         cin >> a[i];
13     }
14     for(int i = 1;i <= n;i++){
15         if(head <= tail && i > k && q[head] == a[i - k]) head ++;
16         while(tail >= head && q[tail] > a[i]) tail --;
17         q[++tail] = a[i];
18         if(i > k - 1){
19             cout << q[head] << " ";
20         }
21     }
22     cout << endl;
23     head = 0, tail = -1;
24     for(int i = 1;i <= n;i++){
25         if(head <= tail && i > k && q[head] == a[i - k]) head ++;
26         while(tail >= head && q[tail] < a[i]) tail --;
27         q[++tail] = a[i];
28         if(i > k - 1){
29             cout << q[head] << " ";
30         }
31     }
32
33 }

```

KMP

- 字符串匹配的一种算法，本质上是根据已有信息减少重复匹配。
- 算法维护一个next数组，存储的是最长的相等真前缀和真后缀的长度。
 - 真前缀：只包含首字母，不包含尾字母的所有子串
 - 真后缀：只包含尾字母，不包含首字母的所有子串
 - 进行next数组初始化时，`i` 是前缀末尾，`j` 是后缀末尾
- 当前面`n`个字符匹配成功，但是第`n + 1`个字符匹配失败，算法将模式串向右滑动`n - next[n]`位，再从第`n+1`位比较

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 1e6 + 10;
5  char s[N],p[N];
6  // 字符串是从下标为1开始，当然也可以从0开始，所以第1个字符能移动的距离ne[1]为0
7  int ne[N];
8  int main(){
9      int n,m;
10     cin >> n >> p + 1 >> m >> s + 1;
11     /*
12     j是存储下标i可以往前移动从而避免重复匹配的下标
13         a b a b a
14         1 2 3 4 5
15     ne  0 0 1 2 3
16         a b a b a
17         a b a b a
18     */
19     for(int i = 2,j = 0;i <= n;i++){
20         while(j && p[i] != p[j + 1]) j = ne[j];
21         if(p[i] == p[j + 1]) j++;
22         ne[i] = j;
23     }
24
25
26     for(int i = 1,j = 0; i <= m;i++){
27         while(j && s[i] != p[j + 1]) j = ne[j];
28         if(s[i] == p[j + 1]) j++;
29         if(j == n) {
30             // 我们代码的下标是从1开始计数，题目要求从0开始，因此不是i-n+1
31             cout << i - n << " ";
32             j = ne[j];
33         }
34     }
35
36 }

```

Trie

高效地存储和查找字符串集合的数据结构

如图所示：

abcdef
 abdef
 aced
 bedf
 bdf
 cdaa

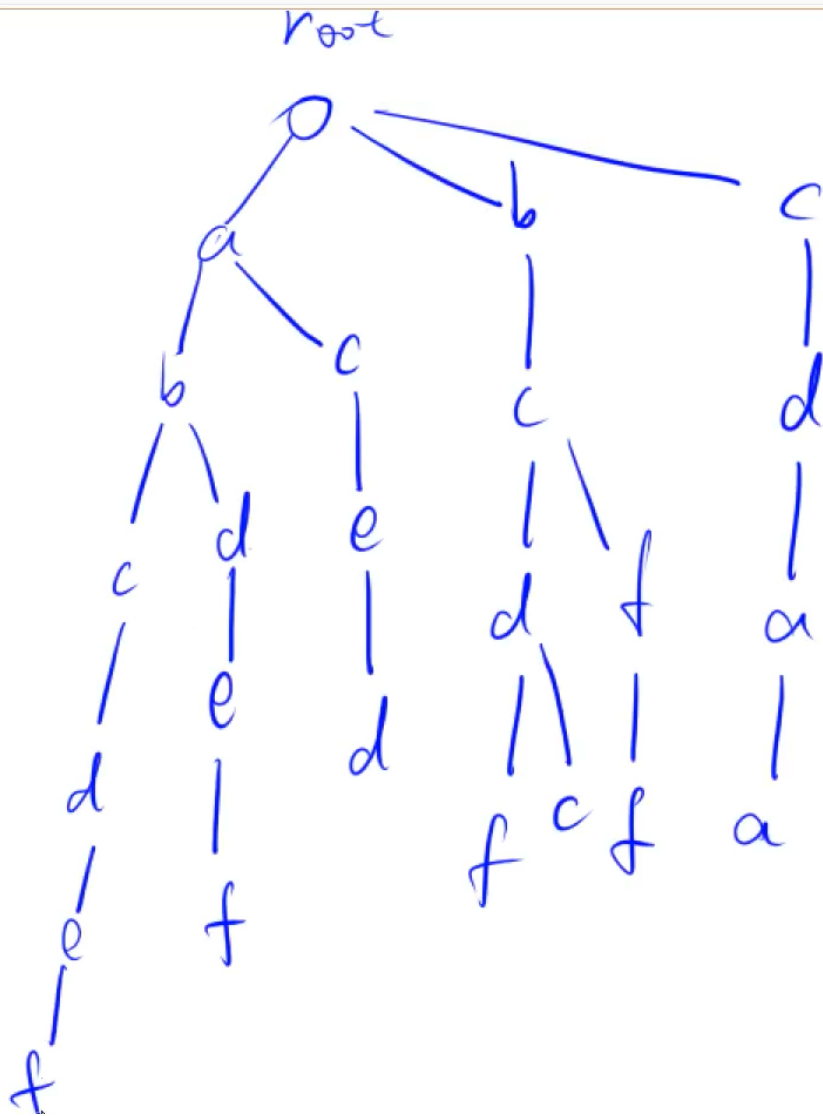


image-20250703102140742

如果某个字符是某个单词结尾，需要在这个字符后打上**标记**

根节点是空节点，下标（编号）为0

```

1  #include<iostream>
2  #include<string.h>
3  using namespace std;
4
5  const int N = 1e5 + 10;
6  int son[N][26],idx,cnt[N];
7  int n;
8
9  void insert(char str[]){
10     int p = 0;
11     for(int i = 0;str[i];i++){
12         if(!son[p][str[i] - 'a']) son[p][str[i] - 'a'] = ++idx;
13         p = son[p][str[i] - 'a'];
14     }
15     cnt[p]++;
16 }
17
18 int query(char str[]){
19
20     int p = 0;
21     for(int i = 0;str[i];i++){
22         if(!son[p][str[i] - 'a']) return 0;
23         p = son[p][str[i] - 'a'];
24     }
25     return cnt[p];
26 }
27
28 int main(){
29     ios::sync_with_stdio(false);
30     cout.tie(0),cin.tie(0);
31     cin >> n;
32     while(n--){
33         string s;
34         char x[N];
35         cin >> s;
36         if(s == "I"){
37             cin >> x;
38             insert(x);
39         }
40         else if(s == "Q"){
41             cin >> x;
42             cout << query(x) << endl;
43         }
44     }
45
46

```

最大异或对

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 1e5 + 10;
5  const int M = 31 * N; //用二进制表示最多有31*N个节点
6  int n;
7  int a[N],son[M][2];
8  int idx = 0;
9  void insert(int x){
10     int p = 0;
11     for(int i = 30;i >= 0;i--){
12         int u = x >> i & 1;
13         if(!son[p][u]) son[p][u] = ++idx;
14         p = son[p][u];
15     }
16 }
17
18 int search(int x){
19     int p = 0,res = 0;
20     for(int i = 30;i >= 0;i--){
21         int u = x >> i & 1;
22         if(son[p][!u]){
23             p = son[p][!u]; // 要找最大的肯定要找不同的, 比如1和0, 0和1
24             res = res * 2 + 1; //比如100 -> 1001 , 起始就是 (100)_2 * 2 + 1 =
(1001)_2 即4 * 2 + 1 = 9
25         }
26         else {
27             p = son[p][u];// 这是因为找不到不同的, 但是为了左移只能退而求其次
28             res = res * 2;
29         }
30     }
31     return res;
32 }
33
34 int main(){
35     scanf("%d",&n);
36     int ans = 0;
37     for(int i = 0;i < n;i++) {
38         scanf("%d",&a[i]);
39         insert(a[i]);
40     }
41     for(int i = 0;i < n;i++) {
42         ans = max(ans,search(a[i]));
43     }
44     cout << ans;

```

并查集

1. 合并两个集合
2. 询问两个元素是否在一个集合中

用树维护集合

- 每一个集合的编号是根节点的编号
- 对于每个节点都存下**父节点是谁**，对于根节点，**它的爸爸是他自己**
- 合并集合：p[x]是x的编号，p[y]是y的编号，只要让p[x] = y即可
- 路径压缩：某一结点第一次向上搜索根节点，找到根节点后**把走过的路径的节点全部指向根节点**

维护额外信息

- 记录集合元素数量：维护一个size数组，这个数组只对根节点有效

合并集合

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 1e5 + 10;
5  int n,m;
6
7  int h[N];
8
9  int find(int x){
10     if(x != h[x]) h[x] = find(h[x]);
11     return h[x];
12 }
13
14
15
16 int main(){
17     ios::sync_with_stdio(false);
18     cin.tie(0),cout.tie(0);
19     cin >> n >> m;
20     for(int i = 0;i < n;i++) h[i] = i; // 每个节点就是一个集合
21     int a,b;
22     while(m--){
23         char c;
24         cin >> c;
25         if(c == 'M'){
26             cin >> a >> b;
27             if(find(a) == find(b)) continue;
28             h[find(a)] = find(b);
29         }
30         else{
31             cin >> a >> b;
32             if(find(a) == find(b)) cout << "Yes" << endl;
33             else cout << "No" << endl;
34         }
35     }
36 }

```

堆

是一个完全二叉树，最后一层节点从左往右排列，其他层节点非空，下标从1开始

用数组模拟，支持修改或者删除任意一个元素（STL里面的不支持）

基本操作

1. 插入一个数
2. 求集合最小值

3. 删除最小值
4. 删除、修改任意一个元素

存储

- x的左儿子: $2x$
- x的右儿子: $2x+1$

向上和向下调整

- 从 $n/2$ 开始调整
- **向上调整** (up) : 用于插入或节点值变小时。
- **向下调整** (down) : 用于删除或节点值变大时。

哈希表

存储结构

1. 开放寻址法

- 只开一个一维数组存储所有哈希值
- 如果位置已经被占, 那么看下一个位置, 循环找到没有人占领的位置
- 核心是find函数
 - 如果x已经存在, 返回x存储的位置
 - 如果x不存在, 返回x要插入的位置

```

1 // 开放寻址法
2 #include<iostream>
3 #include<cstring>
4 using namespace std;
5
6 const int N = 2e5 + 3; // 用两倍是因为要让每个数都能找到位置, +3是经验设定, 可以减少冲突发生的
  概率
7 const int NuLL = 0x3f3f3f3f; //设定初值
8 int h[N];
9 int n;
10
11 // 这个函数能返回x要插入的位置, 也能返回x在哈希表的位置
12 int find(int x){
13     int k = (x % N + N) % N;
14     while(h[k] != NuLL && h[k] != x){
15         k++;
16         if(k == N) k = 0;
17     }
18     return k;
19 }
20
21
22 int main(){
23     memset(h,0x3f,sizeof(h)); // 初始化
24     string s;
25     cin >> n;
26     while(n--){
27         cin >> s;
28         int x;
29         if(s == "I"){
30             cin >> x;
31             h[find(x)] = x;
32         }
33         else if(s == "Q"){
34             cin >> x;
35             int k = find(x);
36             if(h[k] == x) cout << "Yes" << endl;
37             else cout << "No" << endl;
38         }
39     }
40 }

```

2. 拉链法

- 开一个一维数组存所有的哈希值

- 这个一维数组存的是链表的头节点
- 有相同哈希值的数字插到对应的链表里

```

1 // 拉链法
2 #include<iostream>
3 #include<cstring>
4 using namespace std;
5
6 const int N = 1e5 + 3; // 找比100000大的最小质数，可以减少冲突出现的概率
7 int h[N];
8 int e[N],ne[N];
9 int idx = 0;
10 int n ;
11
12 void insert(int x){
13     int k = (x % N + N) % N; //设定哈希函数，这里这么写是因为有负数 先把x压缩到[-N,N]，再
    进一步压缩
14     e[idx] = x;
15     ne[idx] = h[k];
16     h[k] = idx++;
17 }
18
19
20 bool find(int x){
21     int k = (x % N + N) % N;
22     for(int i = h[k];i != -1;i = ne[i]){
23         if(e[i] == x) return true;
24     }
25     return false;
26 }
27
28 int main(){
29     memset(h,-1,sizeof(h)); //初始化哈希表全为-1
30     cin >> n;
31     string s;
32     int x;
33     while(n--){
34         cin >> s;
35         if(s == "I"){
36             cin >> x;
37             insert(x);
38         }
39         else if(s == "Q"){
40             cin >> x;
41             if(find(x)) cout << "Yes" << endl;
42             else cout << "No" << endl;
43         }
44     }
45

```

字符串哈希

用于快速判断两个字符串**是不是相等**

- 把字符串的每个字符都看成一个整数，组合成一个p进制，然后算出这个字符串的十进制数再模一个q，得到这个字符串的哈希值
 - p一般取131或者13331
 - q取 2^{64} ，可以直接用unsigned long long进行存储，溢出也不管，这样就不用写取模
- 会把字符串的每个前缀的哈希值都算出来，用h数组存储
 - `h[0] = 0`
 - `h[i] = h[i - 1] * 131 + str[i];`
- 对于任意子串，可以用这个算法以O(1)复杂度算出来其哈希值
 - $h[L - R] = h[R] - h[L - 1] * 131^{R-L+1}$
 - 131^{R-L+1} 可以用数组求
 - 这是因为在 `h[R]` 中，`h[L-1]` 其实贡献了高位，但是实际 `h[L-1]` 是低位，因此要减去右移后的

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 1e5 + 10;
5  char s[N];
6  unsigned long long P[N],h[N];
7  int n,m;
8  int p = 131;
9
10 int search(int l,int r){
11     return h[r] - h[l - 1] * P[r - l + 1];
12 }
13
14 int main(){
15     cin >> n >> m;
16     cin >> s;
17     P[0] = 1;
18     for(int i = 0;i < n;i++){
19         P[i + 1] = P[i] * p;
20         h[i + 1] = h[i] * p + s[i];
21     }
22
23
24     while(m --){
25         int l1,r1,l2,r2;
26         cin >> l1 >> r1 >> l2 >> r2;
27         if(search(l1,r1) == search(l2,r2)) cout << "Yes" << endl;
28         else cout << "No" << endl;
29     }
30 }

```

STL

vector

- 变长数组，支持比较运算
- 初始化： `vector<int> a = (10,3)` : 初始化长度为10，初值为3的数组
- `size()`: 查看元素个数
- `empty()`: 判空
- `clear()`: 清空
- `front()/back()`: 返回第一个/最后一个数
- `push_back()/pop_back()`: 插入/删除最后一个数
- `begin()/end()`: 第0个数/最后一个数的后面一个数的迭代器

pair<int,int>

- 可以定义一个二元组
- first()/second(): 第1/2个元素
- 初始化: `p = (20, "zxb")` 或者 `p = make_pair(20, "zxb")`

string

- substr(): 返回子串
 - substr(1,2): 从下标为1的字符开始, 输出长度为2的子串
- size(): 查看元素个数
- empty(): 判空
- clear(): 清空

queue

- size(): 查看元素个数
- empty(): 判空
- push(): 往队尾插入
- front(): 返回队头元素
- back(): 返回队尾元素
- pop(): 弹出队头元素

priority_queue (优先队列, 堆)

默认是大根堆

- push(): 插入元素
- front(): 返回堆顶元素
- pop(): 弹出堆顶元素

deque (双端队列)

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- front(): 返回队头元素
- back(): 返回队尾元素
- push_back() / pop_back(): 插入/弹出最后一个数
- push_front() / pop_front(): 插入/弹出第一个数
- begin()/end(): 第0个数/最后一个数的后面一个数的迭代器

stack

- size(): 查看元素个数
- empty(): 判空
- push(): 向栈顶插入元素
- top(): 返回栈顶元素

- pop(): 弹出栈顶元素

set

不能有重复元素

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素
- find(): 查找一个元素
- count(): 返回某个元素个数
- erase()
 - 如果输入是一个数x, 删除所有x
 - 输入一个迭代器, 删除这个迭代器
- lower_bound(): 返回**大于等于**x最小的数的迭代器, 不存在返回end()
- upper_bound(): 返回**大于**x的最小的数的迭代器, 不存在返回end()
- begin()/end(): 第0个数/最后一个数的后面一个数的迭代器

map

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素, 插入的是一个pair
- erase(): 输入的参数是pair或者迭代器
- `m['key']`: 可以用key去索引value
- lower_bound(): 返回**大于等于**x最小的数的迭代器, 不存在返回end()
- upper_bound(): 返回**大于**x的最小的数的迭代器, 不存在返回end()

multiset

可以有重复元素

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素
- find(): 查找一个元素
- count(): 返回某个元素个数
- erase()
 - 如果输入是一个数x, 删除所有x
 - 输入一个迭代器, 删除这个迭代器
- lower_bound(): 返回**大于等于**x最小的数的迭代器, 不存在返回end()
- upper_bound(): 返回**大于**x的最小的数的迭代器, 不存在返回end()

multimap

- size(): 查看元素个数
- empty(): 判空

- `clear()`: 清空
- `insert()`: 插入一个元素, 插入的是一个pair
- `erase()`: 输入的参数是pair或者迭代器
- `m['key']`: 可以用key去索引value
- `lower_bound()`: 返回大于等于x最小的数的迭代器, 不存在返回end()
- `upper_bound()`: 返回大于x的最小的数的迭代器, 不存在返回end()

unordered_map、unordered_set、unordered_multiset、unordered_multimap

基于哈希表实现, 增删改查时间复杂度是 $O(1)$

不支持 `lower_bound()`和`upper_bound()`

搜索和图论

DFS

深度优先搜索, 使用栈

重点: 递归结束条件的选择+状态标记+递归后的恢复

排列数字

```

1  #include<iostream>
2  using namespace std;
3  const int N = 10;
4  int n,t[N];
5  bool state[N];
6
7  void dfs(int p){
8      if(p == n){
9          for(int i = 0;i < n;i++) cout << t[i] << " ";
10         cout << endl;
11     }
12     for(int i = 1;i <= n;i++){
13         if(!state[i]){
14             t[p] = i;
15             state[i] = true;
16             dfs(p + 1);
17             state[i] = false;
18         }
19     }
20 }
21
22
23 int main(){
24     cin >> n;
25     dfs(0);
26 }
27

```

n皇后

```

1  #include<iostream>
2  using namespace std;
3
4  const int N = 12,M = 2 * N;
5  char d[N][N];
6  int n;
7  bool col[N],e[M],ne[M];
8  // 第u行
9  void dfs(int u){
10     if(u == n + 1){
11         for(int i = 1;i <= n;i++){
12             for(int j = 1;j <= n;j++){
13                 cout << d[i][j];
14             }
15             cout << endl;
16         }
17         return ;
18     }
19
20     for(int i = 1;i <= n;i++){
21         if(!col[i] && !e[i + u] && !ne[n - i + u]){
22             col[i] = true,e[i + u] = true,ne[n - i + u] = true;
23             d[u][i] = 'Q';
24             dfs(u + 1);
25             col[i] = false,e[i + u] = false,ne[n - i + u] = false;
26             d[u][i] = '.';
27         }
28     }
29 }
30
31
32 int main(){
33     cin >> n;
34     for(int i = 1;i <= n;i++){
35         for(int j = 1;j <= n;j++){
36             d[i][j] = '.';
37         }
38     }
39     dfs(1);
40     return 0;
}

```

BFS

广度优先搜索，使用队列，一般求什么最短的，最少操作次数的，用BFS

当所有边的权重都为1的时候，才能用BFS

走迷宫

```
1  #include<iostream>
2  using namespace std;
3  typedef pair<int,int> PII;
4  const int N = 110;
5  int d[N][N],p[N][N]; //d存迷宫 p存步数
6  PII q[N * N];
7  bool st[N][N];
8  int head = 0,tail = -1;
9  int dx[4] = {0,1,0,-1},dy[4] = {-1, 0, 1, 0};
10 int n,m;
11 int bfs(){
12     st[0][0] = true;
13     p[0][0] = 0;
14     q[++tail] = {0,0};
15     while(head <= tail){
16         auto x = q[head++];
17         for(int i = 0;i < 4;i++){
18             int a = x.first + dx[i],b = x.second + dy[i]; // a横坐标、b纵坐标
19             if(a >= 0 && a < n && b >= 0 && b < m && !st[a][b] && d[a][b] == 0){
20                 st[a][b] = true;
21                 p[a][b] = p[x.first][x.second] + 1;
22                 q[++tail] = {a,b};
23             }
24         }
25     }
26     return p[n - 1][m - 1];
27 }
28 int main(){
29     cin >> n >> m;
30     for(int i = 0;i < n;i++){
31         for(int j = 0;j < m;j++){
32             cin >> d[i][j];
33         }
34     }
35     cout << bfs();
36 }
```

图中点的层次

```

1  #include<iostream>
2  #include<queue>
3  #include<cstring>
4  using namespace std;
5  const int N = 1e5 + 10;
6  int dist[N],e[N],ne[N],h[N],idx;
7  bool st[N];
8  int n,m;
9  queue<int> q;
10
11 void add(int a,int b){
12     e[idx] = b,ne[idx] = h[a],h[a] = idx ++;
13 }
14
15 int main(){
16     cin >> n >> m;
17     memset(h,-1,sizeof h);
18     memset(dist,0x3f,sizeof(dist));
19     while(m--){
20         int a,b;
21         cin >> a >> b;
22         add(a,b);
23     }
24     q.push(1);
25     dist[1] = 0;
26     st[1] = true;
27     while(q.size()){
28         int t = q.front();
29         q.pop();
30         for(int i = h[t];i != -1;i = ne[i]){
31             int j = e[i];
32             if(!st[j]){
33                 dist[j] = dist[t] + 1;
34                 q.push(j);
35                 st[j] = true;
36             }
37         }
38     }
39     if(dist[n] == 0x3f3f3f3f) cout << -1;
40     else cout << dist[n];
41 }

```

拓扑排序

核心思想是将图中的所有顶点排成一个序列，使得对于图中的每一条有向边($u \rightarrow v$)，在排序中顶点 u 都位于顶点 v 的前面

适用于有向无环图

```

1  #include<iostream>
2  #include<cstring>
3  using namespace std;
4  const int N = 1e5 + 10;
5  int n,m;
6  int h[N],e[N],ne[N],idx;
7  int q[N],d[N];
8
9  void add(int a,int b){
10     e[idx] = b,ne[idx] = h[a],h[a] = idx++;
11 }
12
13 bool topsort(){
14     int head = 0,tail = -1;
15     for(int i = 1;i <= n;i++){
16         if(!d[i]) q[++tail] = i;
17     }
18     while(head <= tail){
19         int x = q[head ++];
20         for(int i = h[x];i != -1;i = ne[i]){
21             int j = e[i];
22             d[j]--;
23             if(!d[j]) q[++tail] = j;
24         }
25     }
26     return tail == n - 1;
27 }
28
29 int main(){
30     memset(h,-1,sizeof h);
31     cin >> n >> m;
32     int a,b;
33     for(int i = 1;i <= m;i++){
34         cin >> a >> b;
35         add(a,b);
36         d[b] ++;
37     }
38     if(topsort()){
39         for(int i = 0;i < n;i++) cout << q[i] << " ";
40     }
41     else {
42         cout << -1;
43     }
44 }
45

```

最短路

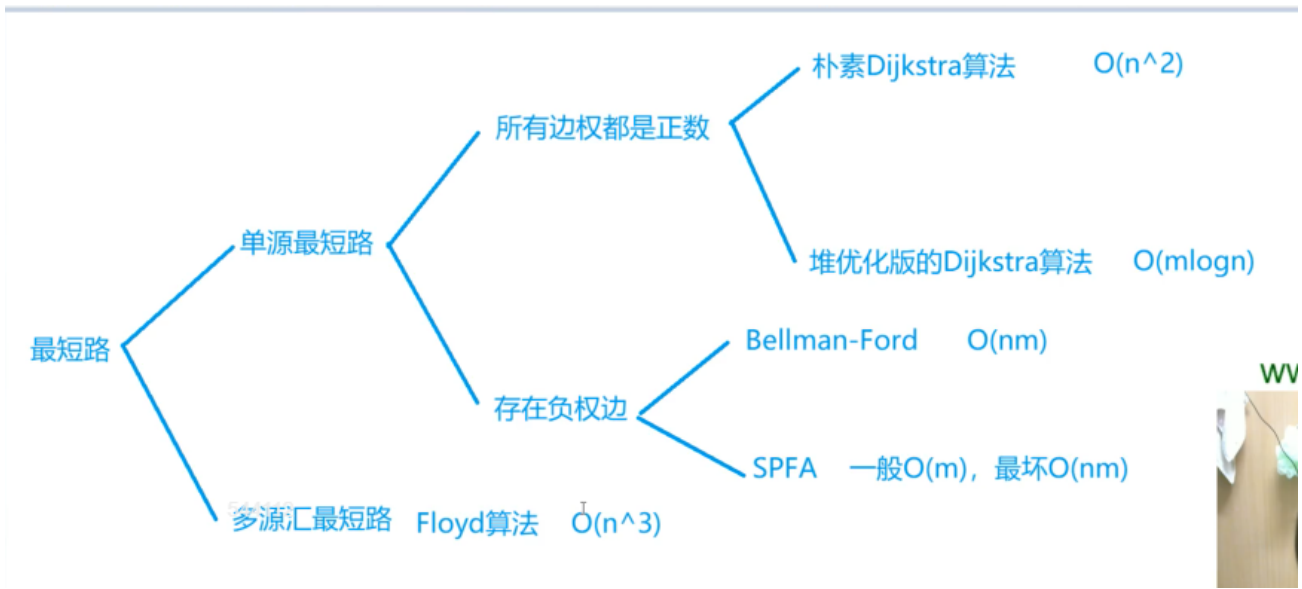


image-20250714111902028

- 朴素Dijkstra算法适用于稠密图
- 堆优化版Dijkstra算法适用于稀疏图：用优先队列
- 稀疏图用邻接表存
- 稠密图用邻接矩阵存
- 没有负环才能用SPFA

朴素Dijkstra算法

- 找到路径最短的点，标记一下
- 用这个点更新其他的路径

```

1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  using namespace std;
5
6  const int N = 510,M = 1e5 + 10;
7  int h[N],e[M],w[M],ne[M],idx;
8  int n,m;
9  int dist[N];
10 bool st[N];
11
12 void add(int a,int b,int c){
13     e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx ++;
14 }
15
16 int dijkstra(){
17     memset(dist,0x3f,sizeof(dist));
18     dist[1] = 0;
19     for(int i = 1;i < n;i++){
20         int t = - 1;
21         for(int j = 1;j <= n;j++){
22             if(!st[j] && (t == -1 || dist[t] > dist[j]) )
23                 t = j;
24         }
25         if (dist[t] == 0x3f3f3f3f) break;
26         st[t] = true;
27         for(int j = h[t]; j != -1;j = ne[j]){ // j为idx
28             int k = e[j]; // k为节点编号
29             dist[k] = min(dist[k],dist[t] + w[j]);
30         }
31     }
32     return dist[n];
33 }
34
35 int main(){
36     cin >> n >> m;
37     memset(h,-1,sizeof h);
38     int a,b,c;
39     while(m--){
40         cin >> a >> b >> c;
41         add(a,b,c);
42     }
43     int res = dijkstra();
44     if(res == 0x3f3f3f3f) cout << -1;
45     else cout << res;

```

堆优化版

- 找到路径最短的点，标记一下（用优先队列直接找到）
- 用这个点更新其他的路径

```

1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  #include<queue>
5  using namespace std;
6
7  const int N = 2e5 + 10,M = 2e5 + 10;
8  int h[N],e[M],w[M],ne[M],idx;
9  int n,m;
10 int dist[N];
11 bool st[N];
12
13 typedef pair<int,int> PII;
14
15 void add(int a,int b,int c){
16     e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx ++;
17 }
18
19 int dijkstra(){
20     memset(dist,0x3f,sizeof(dist));
21     dist[1] = 0;
22     priority_queue <PII,vector<PII>,greater<PII>> heap;
23     heap.push({0,1});
24
25     while(heap.size()){
26         auto x = heap.top();
27         heap.pop();
28         if(st[x.second]) continue;
29         st[x.second] = true;
30         for(int i = h[x.second];i != -1;i = ne[i]){
31             int j = e[i];
32             if(dist[j] > x.first + w[i]){
33                 dist[j] = x.first + w[i];
34                 heap.push({dist[j],j});
35             }
36         }
37
38     }
39     return dist[n];
40 }
41
42 int main(){
43     cin >> n >> m;
44     memset(h,-1,sizeof h);
45     int a,b,c;
46     while(m--){

```

```
47     cin >> a >> b >> c;
48     add(a,b,c);
49 }
50 int res = dijkstra();
51 if(res == 0x3f3f3f3f) cout << -1;
52 else cout << res;
53 }
```

Bellman-ford算法

- 先循环k次（循环k次就是找到不少于k条边的最短路径）
 - 需要拷贝上一次遍历的结果
- 内循环遍历所有的边，更新每个点的最短路径

SPFA算法

- 初始化所有数组
- 当队列不空的时候取出队头元素并更新其他点的路径
- 更新过的点放到队列里面，不能重复放，用st数组判断点是否已经放进队列

```

1  #include<iostream>
2  #include<queue>
3  #include<cstring>
4  using namespace std;
5  const int N = 1e5 + 10;
6  typedef pair<int,int> PII;
7  int n,m;
8
9  int e[N],ne[N],idx,w[N],h[N],dist[N];
10 bool st[N];
11 void add(int x,int y,int z){
12     e[idx] = y,ne[idx] = h[x],w[idx] = z,h[x] = idx ++;
13 }
14
15 int spfa(){
16     memset(dist,0x3f,sizeof dist);
17     queue<PII> q;
18     dist[1] = 0;
19     q.push({0,1}); // 最短距离是0, 编号是1
20     st[1] = true;
21     while(q.size()){
22         auto t = q.front();
23         q.pop();
24         int dis = t.first,vec = t.second;
25         st[vec] = false;
26         for(int i = h[vec];i != -1;i = ne[i]){
27             int j = e[i];
28             if(dist[j] > dist[vec] + w[i]){ // 这里不能用dis, 因为dist[vec]在循环的时
候可能会更新
29                 dist[j] = dist[vec] + w[i];
30                 if(!st[j]){
31                     q.push({dist[j],j});
32                     st[j] = true;
33                 }
34             }
35         }
36     }
37     return dist[n] >= 0x3f3f3f3f / 2; // 可能有负权边
38 }
39
40 int main(){
41     memset(h,-1,sizeof h);
42     cin >> n >> m;
43     while(m--){
44         int x,y,z;
45         cin >> x >> y >> z;

```

```
46     add(x,y,z);
47     }
48     if(spfa()) cout << "impossible" << endl;
49     else cout << dist[n] << endl;
50
```

- 判断负环

- 可以直接维护一个cnt数组，记录最短路径的边数，**当边数大于节点数，说明有负环**
- 需要将所有点都加到队列里面
- dist数组并不是用来记录最短路径的，是一个工具数组，因为只要有负权环，dist存的数就会越来越小，因此其初始值也是任意的

```

1  #include<iostream>
2  #include<queue>
3  #include<cstring>
4  using namespace std;
5  const int N = 2010, M = 10010;
6
7  int n,m;
8
9  int e[M],ne[M],idx,w[M],h[N],dist[N],cnt[N];
10 bool st[N];
11 void add(int x,int y,int z){
12     e[idx] = y,ne[idx] = h[x],w[idx] = z,h[x] = idx ++;
13 }
14
15 bool spfa(){
16     queue<int> q;
17     for(int i = 1;i <= n;i++){
18         q.push(i);
19         st[i] = true;
20     } // 最短距离是0, 编号是1
21     while(q.size()){
22         auto t = q.front();
23         q.pop();
24         st[t] = false;
25         for(int i = h[t];i != -1;i = ne[i]){
26             int j = e[i];
27             if(dist[j] > dist[t] + w[i]){ // 这里不能用dis, 因为dist[vec]在循环的时候
可能会更新
28                 dist[j] = dist[t] + w[i];
29                 cnt[j] = cnt[t] + 1;
30                 if(cnt[j] >= n) return true;
31                 if(!st[j]){
32                     q.push(j);
33                     st[j] = true;
34                 }
35             }
36         }
37     }
38     return false;
39 }
40
41
42 int main(){
43     memset(h,-1,sizeof h);
44     cin >> n >> m;
45     while(m--){

```

```
46     int x,y,z;
47     cin >> x >> y >> z;
48     add(x,y,z);
49 }
50 if(spfa()) cout << "Yes" << endl;
51 else cout << "No" << endl;
52
53 }
```

Floyd算法

- 用邻接矩阵存
- 三层循环，最后邻接矩阵存的就是最短路径长度

```

1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  using namespace std;
5
6  const int N = 210;
7  int n,m,k;
8  int d[N][N];
9
10
11
12 void floyd(){
13     for(int k = 1;k <= n;k ++){
14         for(int i = 1;i <= n;i++){
15             for(int j = 1;j <= n;j++){
16                 d[i][j] = min(d[i][j],d[i][k] + d[k][j]);
17             }
18         }
19     }
20
21 int main(){
22     memset(d,0x3f,sizeof d);
23     cin >> n >> m >> k;
24     for(int i = 1;i <= n;i++) d[i][i] = 0;
25     while(m--){
26         int x,y,z;
27         cin >> x >> y >> z;
28         d[x][y] = min(d[x][y],z);
29     }
30     floyd();
31     while(k--){
32         int x,y;
33         cin >> x >> y;
34         if(d[x][y] >= 0x3f3f3f3f / 2) cout << "impossible" << endl;
35         else cout << d[x][y] << endl;
36     }
37 }

```