NJUPT XCPC Templates

Yunhai Bian

November 26, 2020

Contents

1 数据结构

1.1 树状数组

```
// 注意树状数组不能处理下标0开始
// 一维
int c[N];

inline int lowbit(int x) {
   return x & -x;
}

int add(int x, int y) {
   for (int i = x; i <= n; i += lowbit(i)) c[i] += y;
}

int sum(int x) {
   int res = 0;
   for (int i = x; i; i -= lowbit(i)) res += c[i];
   return res;
}

// 二维
LL c[N][N];

inline int lowbit(int x) {
   return x & -x;
}

void add(int x, int y, LL v) {
   for (int i = x; i <= n;i += lowbit(i))
      for (int j = y; j <= m;j += lowbit(j))
         c[i][j] += v;
}

LL query(int x, int y) {
   LL ans = 0;
   for (int i = x; i; i -= lowbit(i))
      for (int j = y; j; j -= lowbit(j))
         ans += c[i][j];
   return ans;
}
```

1.2 线段树

```
// 例子: 维护区间加法区间求和
#include <bits/stdc++.h>

using namespace std;

typedef long long LL;

```

```cpp
const int N = 100010;

int n, m;
LL a[N];
struct Tree {
   int l, r;
   LL sum, add;
} tr[N << 2];

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

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

void pushdown(int u) { // 先下方标记，再清空标记
   Tree &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
   left.add += root.add, left.sum += (LL)(left.r - left.l + 1) * root.add;
   right.add += root.add, right.sum += (LL)(right.r - right.l + 1) * root.add
       ;
   root.add = 0;
}

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

LL query(int u, int l, int r) {
   if (tr[u].l >= l && tr[u].r <= r) {
      return tr[u].sum;
   } else {
      pushdown(u);
      int mid = tr[u].l + tr[u].r >> 1;
      LL res = 0;
      if (l <= mid) res += query(u << 1, l, r);
```

```
60        if (r > mid) res += query(u << 1 | 1, l, r);
61        return res;
62     }
63  }
64
65  int main() {
66     scanf("%d%d", &n, &m);
67     for (int i = 1; i <= n; i++) scanf("%lld", &a[i]);
68     build(1, 1, n);
69     char op[2];
70     int l, r;
71     while (m--) {
72        scanf("%s%d%d", op, &l, &r);
73        if (*op == 'C') {
74           LL d;
75           scanf("%lld", &d);
76           modify(1, l, r, d);
77        } else {
78           printf("%lld\n", query(1, l, r));
79        }
80     }
81     return 0;
82  }
```

1.3 ST 表

```
1  // ST表可维护区间最值/区间gcd
2  int n, a[N];
3  int f[N][M]; // f[i][j]表示区间[i, i+2^j-1]区间的最大值
4  int Log2[N];
5
6  void ST_pre() {
7     Log2[2] = 1;
8     for (int i = 3; i < N; i++) Log2[i] = Log2[i >> 1] + 1;
9     for (int i = 1; i <= n; i++) f[i][0] = a[i];
10    for (int j = 1; j < M; j++) {
11       for (int i = 1; i + (1 << j) - 1 <= n; i++)
12          f[i][j] = max(f[i][j- 1], f[i + (1 << j - 1)][j - 1]);
13    }
14 }
15
16 int query(int l, int r) {
17    int k = Log2[r - l + 1];
18    return max(f[l][k], f[r - (1 << k) + 1][k]);
19 }
```

1.4 Splay

2 图论

2.1 最短路

2.2 最小生成树

2.3 次小生成树

2.4 有向图的强连通分量

2.5 无向图的双连通分量

2.6 最近公共祖先

2.7 2-SAT

2.8 网络流

2.8.1 Dinic

test

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

using namespace std;

const int N = 100010, M = 200010, INF = 1e9;

int n, m, S, T;
int h[N], e[M], w[M], ne[M], idx;
int q[N];
int d[N], cur[N]; // d[i]表示点i的层次, cur[i]表示i的当前弧

void add(int a, int b, int c) {
   e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
   e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs() { // 判断残留网络是否存在增广路 (即从S到T存在边全大于0的路径)
   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 && w[i]) {
            d[j] = d[t] + 1;
            cur[j] = h[j];
            q[++tt] = j;
            if (j == T) return true; // 找到了增广路, 此时增广路的流量即f[T]
         }
      }
   }
   return false; // 残留网络不存在增广路, 那么此时原图的可行流流量就是最大流
```

```
}

// 从起点到u, 流量最大值为limit
int find(int u, int limit) {
   if (u == T) return limit;
   int flow = 0; // u->T的流量
   for (int i = cur[u]; ~i && flow < limit; i = ne[i]) {
      int j = e[i];
      cur[u] = i; // 更新当前弧
      if (d[j] == d[u] + 1 && w[i]) {
         int t = find(j, min(w[i], limit - flow));
         if (!t) d[j] = -1; // 删点
         w[i] -= t, w[i ^ 1] += t, flow += t;
      }
   }
   return flow;
}

int dinic() {
   int max_flow = 0;
   while (bfs()) while (int flow = find(S, INF)) max_flow += flow;
   return max_flow;
}

int main() {
   cin >> n >> m >> S >> T;
   memset(h, -1, sizeof h);
   while (m--) {
      int a, b, c;
      scanf("%d%d%d", &a, &b, &c);
      add(a, b, c); // 初始流量为0, 对于原图的残留网络, 正向边容量为c-0=c, 反向边容量
          为0+0=0
   }
   printf("%d\n", dinic());
   return 0;
}
```

2.8.2 EK

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

using namespace std;

const int N = 1010, M = 20010, INF = 1e9;

int n, m, S, T;
int h[N], e[M], w[M], ne[M], idx;
bool st[N];
int q[N], f[N]; // f[i]表示以i结尾的增广路径的流量 (即路径上容量的最小值)
int pre[N]; // pre[i]表示i的前驱边的编号 (即指向i的边)

void add(int a, int b, int c) {
```

```
14    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
15    e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
16 }
17
18 bool bfs() { // 判断残留网络是否存在增广路 (即从S到T存在边全大于0的路径)
19    int hh = 0, tt = -1;
20    memset(st, false, sizeof st);
21    q[++tt] = S, st[S] = true, f[S] = INF;
22    while (hh <= tt) {
23       int t = q[hh++];
24       for (int i = h[t]; ~i; i = ne[i]) {
25          int j = e[i];
26          if (!st[j] && w[i]) {
27             q[++tt] = j;
28             st[j] = true;
29             pre[j] = i; // 记录j的前驱边
30             f[j] = min(f[t], w[i]);
31             if (j == T) return true; // j是终点, 即找到了增广路, 此时增广路的流量即f
                   [T]
32          }
33       }
34    }
35    return false; // 残留网络不存在增广路, 那么此时原图的可行流流量就是最大流
36 }
37
38 int EK() {
39    int flow = 0;
40    while (bfs()) { // 如果残留网络存在增广路f', 就将他的流量加到原网络的流量上
41       flow += f[T];
42       for (int i = T; i != S; i = e[pre[i] ^ 1]) { // 更新残留网络
43          w[pre[i]] -= f[T], w[pre[i] ^ 1] += f[T];
44       }
45    }
46    return flow; // 最大流
47 }
48
49 int main() {
50    cin >> n >> m >> S >> T;
51    memset(h, -1, sizeof h);
52    while (m--) {
53       int a, b, c;
54       scanf("%d%d%d", &a, &b, &c);
55       add(a, b, c); // 初始流量为0, 对于原图的残留网络, 正向边容量为c-0=c, 反向边容量
             为0+0=0
56    }
57    printf("%d\n", EK());
58    return 0;
59 }
```

2.9 二分图

3 其他

3.1 高精度

```
struct hll {
   int num[4010], len, sign;
   hll() { len = 0, sign = 1; }
   hll(int x) { *this = x; }
   hll(long long x) { *this = x; }
   hll(char *ss) { *this = ss; }
   hll(string ss) { *this = ss; }
   hll& operator = (const int &x) {
      int val = x;
      if (val >= 0) sign = 1, len = 0;
      else if (val < 0) sign = -1, val = -val, len = 0;
      do {
         num[len++] = val % 10, val /= 10;
      } while (val);
      return *this;
   }
   hll& operator = (const long long &x) {
      long long val = x;
      if (val >= 0) sign = 1, len = 0;
      else if (val < 0) sign = -1, val = -val, len = 0;
      do {
         num[len++] = val % 10, val /= 10;
      } while (val);
      return *this;
   }
   hll& operator = (const string &ss) {
      len = ss.size();
      int start;
      if (ss[0] == '-') sign = -1, start = 1;
      else sign = 1, start = 0;
      for (int i = len - 1; i >= start; i--) num[len - i - 1] = ss[i] - '0';
      if (sign == -1) len--;
      return *this;
   }
   hll& operator = (const char *ss) {
      len = strlen(ss);
      int start;
      if (ss[0] == '-') sign = -1, start = 1;
      else sign = 1, start = 0;
      for (int i = len - 1; i >= start; i--) num[len - i - 1] = ss[i] - '0';
      if (sign == -1) len--;
      return *this;
   }
   hll& operator = (const hll &t) {
      len = t.len, sign = t.sign;
      for (int i = 0; i < len; i++) num[i] = t.num[i];
      return *this;
```

```
48    }
49    int abs_cmp(const hll &a, const hll &b) const { // |a|>|b|时返回1，相等返回0
          ，小于返回-1
50       if (a.len > b.len) return 1;
51       else if (a.len < b.len) return -1;
52       else {
53          for (int i = a.len - 1; i >= 0; i--) {
54             if (a.num[i] < b.num[i]) return -1;
55             if (a.num[i] > b.num[i]) return 1;
56          }
57          return 0;
58       }
59    }
60    int cmp(const hll &t) const { // *this与t比较，小于返回-1，等于返回0，大于返回1
61       if (sign != t.sign) {
62          if (sign == 1) return 1;
63          else return -1;
64       } else {
65          if (abs_cmp(*this, t) == 1) return sign;
66          else if (abs_cmp(*this, t) == 0) return 0;
67          else return -sign;
68       }
69    }
70    hll abs_plus(const hll &a, const hll &b) { // |a|+|b|，ans的符号与a和b原来的
          符号相同
71       hll ans;
72       ans.sign = a.sign;
73       for (int i = 0, carry = 0; i < a.len || i < b.len || carry; i++) {
74          if (i < a.len) carry += a.num[i];
75          if (i < b.len) carry += b.num[i];
76          ans.num[ans.len++] = carry % 10;
77          carry /= 10;
78       }
79       return ans;
80    }
81    hll abs_minus(const hll &a, const hll &b) { // ||a|-|b||，ans的符号为|a|-|b
          |的符号
82       hll ans, c, d;
83       if (abs_cmp(a, b) >= 0) ans.sign = 1, c = a, d = b;
84       else ans.sign = -1, c = b, d = a;
85       for (int i = 0, borrow = 0; i < c.len; i++) {
86          borrow = c.num[i] - borrow;
87          if (i < d.len) borrow -= d.num[i];
88          ans.num[ans.len++] = (borrow + 10) % 10;
89          if (borrow >= 0) borrow = 0;
90          else borrow = 1;
91       }
92       while (ans.len > 1 && ans.num[ans.len - 1] == 0) ans.len--; // 去除前导0
93       return ans;
94    }
95    bool operator == (const hll &t) const { return cmp(t) == 0; }
96    bool operator != (const hll &t) const { return !(cmp(t) == 0); }
97    bool operator < (const hll &t) const { return cmp(t) == -1; }
```

```
98    bool operator > (const hll &t) const { return cmp(t) == 1; }
99    bool operator <= (const hll &t) const { return !(cmp(t) == 1); }
100   bool operator >= (const hll &t) const { return !(cmp(t) == -1); }
101   hll operator + (const hll &t) {
102      hll ans;
103      if (sign == t.sign) { // 同号 直接相加, 符号不变
104         ans = abs_plus(*this, t);
105      } else { // 异号
106         if (sign == 1) { // 前正 + 后负 == 前绝对值 - 后绝对值
107            ans = abs_minus(*this, t);
108         } else { // 前负 + 后正 == 后绝对值 - 前绝对值
109            ans = abs_minus(t, *this);
110         }
111      }
112      return ans;
113   }
114   hll operator - (const hll &t) {
115      hll ans;
116      if (sign == t.sign) { // 同号
117         ans = abs_minus(*this, t);
118         if (sign == 1) { // 前正 - 后正
119            ; // 不用做了
120         } else { // 前负 - 后负
121            ans.sign *= -1;
122         }
123      } else { // 异号
124         if (sign == 1) { // 前正 - 后负 == 前绝对值 + 后绝对值
125            ans = abs_plus(*this, t);
126         } else { // 前负 - 后正 == -(前绝对值 + 前绝对值)
127            ans = abs_plus(t, *this);
128            ans.sign = -1;
129         }
130      }
131      return ans;
132   }
133   hll operator * (const hll &t) { // 高精度*高精度
134      hll ans;
135      memset(ans.num, 0, len + t.len << 2);
136      ans.sign = sign * t.sign;
137      ans.len = len + t.len - 1; // a位数乘以b位数, 得到的结果是a+b-1位数, 或a+b位
          数
138      for (int i = 0; i < len; i++) {
139         for (int j = 0; j < t.len; j++)
140            ans.num[i + j] += num[i] * t.num[j];
141      }
142      for (int i = 0; i < ans.len - 1; i++) {
143         if (ans.num[i] >= 10) {
144            ans.num[i + 1] += ans.num[i] / 10;
145            ans.num[i] %= 10;
146         }
147      }
148      // 看最高位是否需要进位, 如果有进位, 答案最终是a+b位数, 否则是a+b-1位数
149      if (ans.num[ans.len - 1] >= 10) {
```

```
150             ans.num[ans.len] = ans.num[ans.len - 1] / 10;
151             ans.num[ans.len - 1] %= 10;
152             ans.len++;
153         }
154         while (ans.len > 1 && ans.num[ans.len - 1] == 0) ans.len--; // 去除前导0
155         return ans;
156     }
157     hll operator += (const hll &t) {
158         return *this + t;
159     }
160     hll operator *= (const hll &t) {
161         return *this * t;
162     }
163     void print() {
164         if (sign == -1) putchar('-');
165         for (int i = len - 1; i >= 0; i--) putchar(num[i] + '0');
166     }
167 };
```

3.2 莫队