基础数据结构入门

丁聪

计算机试验班61

2017年6月

前言

- 由于大家大部分有数据结构课
- 链表、栈、队列到时候会讲
- 所以今天就不用讲那些东西了
- 今天主要讲一些维护序列的技巧
- 以及一些数据结构课上不会讲的东西
- 上过数据结构课的同学应该也会有所收获

Tips

- 听课过程中有这些问题可以提问：
  - 厕所在哪里
- 这些问题就别问了
  - 听不懂
  - 听不懂
  - 听不懂
- 这些可以自行解决
  - 为什么我打不开4399
  - 为什么我电脑自动跳出了游戏
- 哦好像打反了，不要在意这些细节

例题0

- 给定一个长度为n的序列$a_1, a_2, ..., a_n$
- 给出q个询问，形如$x\ y$
- 对于每个询问，输出$\sum_{i=x}^{y} a_i$
- $n = 1000, q = 1000$

解法

- 按要求来就行了，没啥好说的
- 代码:

```
cin >> n >> q;
for(int i = 1; i <= n; i++)
    cin >> a[i];
while(q--) {
    int x, y, res = 0;
    cin >> x >> y;
    for(int i = x; i <= y; i++)
        res += a[i];
    cout << res << endl;
}
```

- 时间复杂度$O(nq)$

例题1

- 给定一个长度为n的序列a_1, a_2, ..., a_n
- 给出q个询问，形如x y
- 对于每个询问，输出$\sum_{i=x}^{y} a_i$
- $n = 10^5$, $q = 10^5$

解法

- 由于$n \times q = 10^{10}$, 所以刚才$O(nq)$ 的解法显然不行
- 引入一个新序列sum_1, sum_2, ..., sum_n
- 其中$sum_i = \sum_{k=1}^{i} a_k$, 特别的, 规定$sum_0 = 0$
- 容易发现$\sum_{i=x}^{y} a_i = sum_y - sum_{x-1}$
- 这样就可以在$O(1)$时间内回答一个询问了
- 如何求出sum_i ?
- 显然$sum_i = sum_{i-1} + a_i$

解法

- 代码:

```
cin >> n >> q;
for(int i = 1; i <= n; i++) {
    cin >> a[i];
    sum[i] = sum[i - 1] + a[i];
}
while(q--) {
    int x, y; cin >> x >> y;
    cout << sum[y] - sum[x - 1] << endl;
}
```

- 时间复杂度$O(n+q)$

例题2

- 给定一个长度为n的序列$a_1, a_2, ..., a_n$
- 给出q个操作，形如$x\ y\ z$
- 对于每个操作，将下标介于$[x, y]$之间的元素加上z
- 输出经过操作后的序列
- $n = 10^5$, $q = 10^5$

解法

- 由于$n \times q = 10^{10}$, 所以直接模拟是不行的
- 引入一个新序列b_1, b_2, ..., b_n
- 其中$b_i = a_i - a_{i-1}$, 特别的, 认为$a_0 = 0$
- 下面观察一下进行修改操作时, 两个序列的变化

解法

- 比如初始的序列a为3 1 4 1 5 9 2 6
- 现在对其下标介于$[2,5]$之间的数加上6
- 新序列变为3 7 10 7 11 9 2 6
- 分别求出这两个序列的b序列，为:
- 3 -2 3 -3 4 4 -7 4
- 3 4 3 -3 4 -2 -7 4
- 只有第2项和第6项发生了变化
- 第2项增大了6，第6项减小了6

解法

- 闭上眼睛想一下可以发现:
- 对于一次操作$x\ y\ z$
- 序列a对应的序列b只有两项发生变化
- 具体来说，即$b_x = b_x + z$，$b_{y+1} = b_{y+1} - z$
- 也就是说对于一次修改，序列b的变化是$O(1)$的
- 所以可以把操作作用到序列b上，最后再由序列b得到序列a
- 如何由序列b得到序列a？
- 显然$a_i = a_{i-1} + b_i$

解法

- 代码:

```
cin >> n >> q;
for(int i = 1; i <= n; i++) {
    cin >> a[i];
    b[i] = a[i] - a[i - 1];
}
while(q--) {
    int x, y, z;
    cin >> x >> y >> z;
    b[x] += z, b[y + 1] -= z;
}
for(int i = 1; i <= n; i++) {
    a[i] = a[i - 1] + b[i];
    cout << a[i] << endl;
}
```

- 时间复杂度$O(n+q)$

例题3

- 给定一个长度为n的序列$a_1, a_2, ..., a_n$
- 给出q个操作，操作分为两种
- 对于形如1 x y的操作，将下标为x的元素加上y
- 对于形如2 x y的操作，输出$\sum_{i=x}^{y} a_i$
- $n = 10^5$, $q = 10^5$

解法1

- 由于$n \times q = 10^{10}$，所以直接模拟是不行的
- 如果使用前缀和，修改的复杂度是$O(n)$的，不能接受
- 现在将序列均匀分成若干块，除最后一块外每块s个元素
- 预处理出每块所有元素的和，形成一个新序列，记为序列c
- 对于一个区间$[x, y]$，容易发现其可以覆盖$O(\frac{n}{s})$个整块，两边剩下$O(s)$零碎的部分
- 所以对于单次查询区间和，可以以$O(s + \frac{n}{s})$的复杂度解决
- 根据基本不等式，$s = \sqrt{n}$时上式取最小，为$O(\sqrt{n})$
- 对于修改，进行如下操作(假设第x个元素属于第t块):
- $a_x = a_x + y$, $c_t = c_t + y$
- 显然可以符合题意
- 总时间复杂度$O(q\sqrt{n})$

解法1

- 比如初始的序列a为3 1 4 1 5 9 2 6
- 取$s=2$, 原序列被分成了[3 1] [4 1] [5 9] [2 6]一共4块
- 对应的序列c为4 5 14 8
- 对于一次查询, 比如$[2,7]$, 这个区间可以分成两部分看:
- 一部分是两边属于第一块和第四块的零碎部分, a_2和a_7
- 还有一部分是中间覆盖的第二块和第三块这两整块
- 对于零碎部分直接暴力处理, 整块部分通过序列c快速得到答案
- 所以对于这个询问, 答案就是$a_1+c_2+c_3+a_7=22$

解法1

- 求出序列c

```
cin >> n >> q;
int s = (int) sqrt(n);
for(int i = 1; i <= n; i++) {
    cin >> a[i];
    belong[i] = (i - 1) / s + 1;
    c[belong[i]] += a[i];
}
for(int i = 1; i <= s; i++)
    startPos[i] = (i - 1) * s + 1;
```

解法1

- 修改操作

```
while(q--) {
    int opt, x, y;
    cin >> opt >> x >> y;
    if(opt == 1) {
        a[x] += y;
        c[belong[x]] += y;
    }
```

解法1

- 查询操作

```
else if(opt == 2) {
    int b1 = belong[x];
    int b2 = belong[y];
    if(b1 == b2) {
        for(int i = x; i <= y; i++)
            res += a[i];
        cout << res << endl;
    }
```

- 查询操作

```
        else if(b1 != b2) {
            for(int i = b1 + 1; i <= b2 - 1; i++)
                res += c[i];
            for(int i = x; i < startPos[b1 + 1]; i++)
                res += a[i];
            for(int i = y; i >= startPos[b2]; i--)
                res += a[i];
            cout << res << endl;
        }
    }
}
```

解法2

- 刚才是把序列分块，来加快查询
- 其实也可以把查询分块，加快修改
- 首先考虑一种不那么暴力的暴力
- 对于每次修改操作，只是记录下来，不真的修改
- 查询的时候，首先通过前缀和得到无修改时的答案
- 再遍历所有修改，加上对查询元素有影响的修改的贡献

解法2

- 代码

```
cin >> n >> q;
for(int i = 1; i <= n; i++) {
    cin >> a[i];
    sum[i] = sum[i - 1] + a[i];
}
while(q--) {
    int opt, x, y;
    cin >> opt >> x >> y;
    if(opt == 1) {
        modifyPos[++tot] = x;
        modifyVal[tot] = y;
    }
```

解法2

- ```
      else if(opt == 2) {
          int res = sum[y] - sum[x - 1];
          for(int i = 1; i <= tot; i++)
              if(x <= modifyPos[i] && y >= modifyPos[i])
                  res += modifyVal[i];
          cout << res << endl;
      }
  }
  ```
- 时间复杂度$O(q^2)$
  ```

解法2

- 可以发现，对于靠前的询问，处理很快，越往后的询问处理越慢
- 为了解决这个问题，可以定期重构序列a和sum
- 重构就是根据当前的修改和初始序列，得到新的一个序列
- 假设每隔s个修改就重构序列，一共要进行$O(\frac{q}{s})$次重构
- 这样可以保证任意时刻，有效的修改是$O(s)$个
- 所以对于一个查询，只需要往前找$O(s)$个修改即可
- 因此查询的总复杂度是$O(qs)$
- 重构操作只需要将最近的$O(s)$个修改作用到序列a上
- 再$O(n)$求一遍前缀和即可，故复杂度$O(\frac{q}{s} \times n)$
- 总时间复杂度为$O(qs + \frac{nq}{s})$
- 令$s = \sqrt{n}$，上式取得最小，为$O(q\sqrt{n})$

解法2

- 代码

```
cin >> n >> q;
for(int i = 1; i <= n; i++) {
    cin >> a[i];
    sum[i] = sum[i - 1] + a[i];
}
int lastModify = 1;
int s = (int) sqrt(n);
```

解法2

- ```
  while(q--) {
      int opt, x, y;
      cin >> opt >> x >> y;
      if(opt == 1) {
          modifyPos[++tot] = x;
          modifyVal[tot] = y;
          if(tot == s + lastModify) {
              for(int i = lastModify; i <= tot; i++)
                  a[modifyPos[i]] += modifyVal[i];
              for(int i = 1; i <= n; i++)
                  sum[i] = sum[i - 1] + a[i];
              lastModify = tot + 1;
          }
      }
  ```
  ```

解法2

- ```
      else if(opt == 2) {
          int res = sum[y] - sum[x - 1];
          for(int i = lastModify; i <= tot; i++)
              if(x <= modifyPos[i] && y >= modifyPos[i])
                  res += modifyVal[i];
          cout << res << endl;
      }
  }
  ```
  ```

例题4

- 给定一个长度为n的序列a_1, a_2, ..., a_n
- 给出q个操作，操作分为两种
- 对于形如$1\ x\ y\ z$的操作，将下标介于$[x, y]$的元素加上z
- 对于形如$2\ x$的操作，输出a_x
- $n = 10^5$, $q = 10^5$

解法

- 先求出序列a的差分序列b，$b_i = a_i - a_{i-1}$
- 容易得到$a_i = \sum_{k=1}^{i} b_k$
- 所以对于操作2，答案就是$\sum_{k=1}^{x} b_k$
- 对于操作1，只需要$b_x = b_x + z$，$b_{y+1} = b_{y+1} - z$
- 这样问题就转化成了单点修改，区间和查询了
- 和上一道题相同，可以在$O(q\sqrt{n})$时间内解决
- 留作作业

例题5

- 给定一个$n \times m$的矩形，其中第i行第j列的值为$a_{i,j}$
- 给出q个询问，形如x_1 y_1 x_2 y_2
- 对于每个询问，输出$\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} a_{i,j}$
- $n = 1000$, $m = 1000$, $q = 10^5$

- 对于一维的情况，求出前缀和就可以快速回答询问了
- 对于二维的情况，要求出二维前缀和
- 引入一个新序列$\{sum_{n,m}\}$，其中$sum_{i,j} = \sum_{s=1}^{i} \sum_{t=1}^{j} a_{s,t}$
- 对于一个询问，答案可以表示为$sum_{x_2,y_2} + sum_{x_1,y_1} - sum_{x_1,y_2} - sum_{x_2,y_1}$
- 这样就可以$O(1)$回答询问了
- 如何得到序列$\{sum_{n,m}\}$？
- $sum_{i,j} = sum_{i-1,j} + sum_{i,j-1} - sum_{i-1,j-1}$
- 时间复杂度$O(nm + q)$

解法

- 代码:

```
cin >> n >> m >> q;
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++) {
        cin >> a[i][j];
        sum[i][j] = sum[i-1][j] + sum[i][j-1];
        sum[i][j] += a[i][j] - sum[i-1][j-1];
    }
while(q--) {
    int x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;
    int res = sum[x2][y2] + sum[x1][y1];
    res -= sum[x1][y2] + sum[x2][y1];
    cout<< res << endl;
}
```

例题6

- 给定一个$n \times m$的矩形，其中第i行第j列的值为$a_{i,j}$
- 给出q个操作，操作有两种
- 对于形如$1\ x_1\ y_1\ x_2\ y_2$的操作，输出$\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} a_{i,j}$
- 对于形如$2\ x\ y\ z$的操作，将$a_{x,y}$加上z
- $n = 1000$，$m = 1000$，$q = 10^5$

解法

- 仿照一维的情况，这个问题可以分块解决
- 二维的情况下，序列分块情况复杂不方便操作
- 所以一般使用询问分块的方法
- 具体实现和一维情况基本一样
- 首先得到二维前缀和序列$\{sum_{n,m}\}$
- 每进行$O(s)$次操作就重构前缀和序列
- 每次询问在前缀和序列的基础上加上最近$O(s)$次修改的贡献
- 总时间复杂度$O(qs + \frac{q}{s} \times nm)$
- 令$s = \sqrt{nm}$，上式取得最小，为$O(q\sqrt{nm})$

解法

- 代码:

```
cin >> n >> m >> q;
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
        cin >> a[i][j];
int s = (int) sqrt(n * m);
for(int i = 1; i <= n; i++) {
    sum[i][j] = sum[i - 1][j] + sum[i][j - 1];
    sum[i][j] = sum[i][j] - sum[i - 1][j - 1];
    sum[i][j] = sum[i][j] + a[i][j];
}
```

解法

- ```
  while(q--) {
      int opt; cin >> opt;
      if(opt == 1) {
          int x1, y1, x2, y2;
          cin >> x1 >> y1 >> x2 >> y2;
          int res = sum[x2][y2] + sum[x1][y1];
          res -= sum[x1][y2] - sum[x2][y1];
          for(int i = lastModify; i <= tot; i++)
              if(modifyX[i] >= x1 && modifyX[i] <= x2)
                  if(modifyY[i] >= y1 && modifyY[i] <= y2)
                      res += modifyZ[i];
          cout << res << endl;
      }
  ```
  ```

解法

- ```
  else if(opt == 2) {
      cin >> modifyX[++tot] >> modifyY[tot] >> modifyZ[tot];
      if(tot == lastPos + s) {
          for(int i = lastPos; i <= tot; i++)
              a[modifyX[i]][modifyY[i]] += modifyZ[i];
          for(int i = 1; i <= n; i++)
              for(int j = 1; j <= m; j++) {
                  sum[i][j] = sum[i-1][j] + sum[i][j-1];
                  sum[i][j] += a[i][j] - sum[i-1][j-1];
              }
          lastPos = tot;
      }
  }
  ```
  ```

例题7

- 给定一个$n \times m$的矩形，其中第i行第j列的值为$a_{i,j}$
- 给出q个操作，操作有两种
- 对于形如1 x_1 y_1 x_2 y_2 z 的操作，将(x_1, y_1)-(x_2, y_2)这段矩形区域的所有元素加上z
- 对于形如2 x y的操作，输出$a_{x,y}$
- $n = 1000$, $m = 1000$, $q = 10^5$

解法

- 由一维的区间加单点求和的解法可以想到
- 需要构造一个新序列$\{b_{n,m}\}$，使得$a_{i,j}=\sum_{s=1}^{i}\sum_{t=1}^{j}b_{s,t}$
- 经过一番试验，可以发现$b_{i,j}=a_{i,j}+a_{i-1,j-1}-a_{i,j-1}-a_{i-1,j}$
- 这样操作1就变成了$b_{x_1,y_1}=b_{x_1,y_1}+z$，$b_{x_2+1,y_2+1}=b_{x_2+1,y_2+1}+z$，$b_{x_1,y_2+1}=b_{x_1,y_2+1}-z$，$b_{x_2+1,y_1}=b_{x_2+1,y_1}-z$
- 操作2就变成了求$\sum_{i=1}^{x}\sum_{j=1}^{y}b_{i,j}$
- 转化成了上一个问题，可以在$O(q\sqrt{nm})$时间内解决
- 留作作业

例题8

- 给定一个长度为n的序列$a_1, a_2, ..., a_n$
- 给出q个操作，操作分为两种
- 对于形如$1\ x\ y$的操作，将下标为x的元素加上y
- 对于形如$2\ x\ y$的操作，输出$\sum_{i=x}^{y} a_i$
- $n = 10^6$, $q = 10^6$

解法

- 这道题的题意和例题3是一样的，只是数据范围乘了10
- 对于10^6的数据，$O(q\sqrt{n})$的做法是难以在规定时间内通过的
- 所以需要一个更加高效的结构来维护这个序列

树状数组(Binary Index Tree)

- 构造一个序列c，$c_i = \sum_{j=i-2^k+1}^{i} a_j$，其中$k$ 表示i的二进制表示中末尾连续0的个数，记$lowbit(i) = 2^k$，特别的，规定$lowbit(0) = 0$
- 例如$i = 11000_{(2)}$，则$k = 3$，$lowbit(i) = 2^3 = 8$
- 如何求$lowbit(i)$？
- $lowbit(i) = i \ \& \ -i$

树状数组(Binary Index Tree)

- 为什么要求$lowbit(i)$？
- 因为其有一些神奇的性质
- 性质一：$lowbit(x \pm lowbit(x)) \geq 2 \times lowbit(x)$, $(x - lowbit(x) \neq 0)$
- 证明：对于x，设其二进制表示为$\overline{x_d x_{d-1} \ldots x_2 x_1 x_0}$
- 设其末尾第一个不为0的位为x_p，则对于$\forall i \in [0, p)$，有$x_i = 0$
- 由$lowbit$的定义可知，$lowbit(x)$的二进制表示为$\overline{1\underbrace{00\ldots0}_{p}}$
- 所以$x - lowbit(x)$的二进制表示为$\overline{x_d x_{d-1} \ldots x_{p+1}\underbrace{00\ldots0}_{p+1}}$
- 因为$x - lowbit(x) \neq 0$，由$lowbit$的定义可知$lowbit(x - lowbit(x)) \geq 2^{p+1} = 2 \times 2^p = 2 \times lowbit(x)$
- 同理也可证加法成立，得证

树状数组(Binary Index Tree)

- 性质二：如果$x - lowbit(x) = k(k \neq 0)$，则对$\forall y \in [x+1, x+lowbit(x)-1]$，有$y - lowbit(y) > k$，除此之外，当$y = x + lowbit(x)$时，有$y - lowbit(y) < k$
- 证明：由上个性质的证明可以知道
- 对于x，设其二进制表示为$\overline{x_d x_{d-1} \ldots x_2 x_1 x_0}$
- 设其末尾第一个不为0的位为x_p，那么$x - lowbit(x)$的二进制表示为$\overline{x_d x_{d-1} \ldots x_{p+1} \underbrace{00 \ldots 0}_{p+1}}$
- 又因为对$\forall i \in [0, p)$，有$x_i = 0$，所以$x - lowbit(x)$的二进制也可以表示为$x_d x_{d-1} \ldots x_{d+1} 0 x_{d-1} \ldots x_1 x_0$
- 所以$x - lowbit(x)$其实就是把x的二进制的末尾第一个1变成了0
- 之后再通过二进制意义下的讨论，容易证明该性质

解法

- 现在考虑如何使用树状数组解决本题
- 显然，对a_x进行修改时，所影响的在序列c中下标最小的元素为c_x
- 由性质二可知，继c_x之后，影响的下一个元素为$c_{x+lowbit(x)}$
- 如此循环下去，直到下标至n，修改操作完成
- 在进行区间和查询的时候，为了方便操作，可以先求出$\sum_{i=1}^{x-1} a_i$和$\sum_{i=1}^{y} a_i$，再作差
- 如何通过树状数组求出$\sum_{i=1}^{x} a_i$？
- 显然$\sum_{i=1}^{x} a_i = c_x + \sum_{i=1}^{x-lowbit(x)} a_i$
- 如此循环下去，直到下标至1，查询操作完成

解法

- 时间复杂度如何?
- 修改和查询操作，都是由一个x，不断的变成$x + lowbit(x)$或$x - lowbit(x)$来完成的
- 由性质一可得，每次变化的变化量都不小于上一次变化量的二倍
- 假设第一次变化量为l_0，则$l_1 \geq 2l_0$，$l_2 \geq 2l_1$
- 设一共变化了m次，则容易发现$O(2^m) = O(n)$，即$O(m) = O(\log n)$
- 所以每次的变化次数是$O(\log n)$级别的
- 也就是说对于一次修改或查询，时间复杂度是$O(\log n)$的
- 故总复杂度$O(q \log n)$

解法

- 代码:

```
cin >> n >> q;
for(int i = 1; i <= n; i++)
    cin >> a[i];
for(int i = 1; i <= n; i++)
    for(int j = i; j >= i - lowbit(i) + 1; j--)
        c[i] += a[j];
```

解法

- ```
  while(q--) {
      int opt, x, y;
      cin >> opt >> x >> y;
      if(opt == 1) {
          int idx = x;
          while(idx <= n) {
              c[idx] += y;
              idx += lowbit(idx);
          }
      }
  ```
  ```

解法

- ```
      else if(opt == 2) {
          int idx = y;
          int res = 0;
          while(idx >= 1) {
              res += c[idx];
              idx -= lowbit(idx);
          }
          idx = x - 1;
          while(idx >= 1) {
              res -= c[idx];
              idx -= lowbit(idx);
          }
          cout << res << endl;
      }
  }
  ```
  ```

例题9

- 给定一个长度为n的序列a_1, a_2, ..., a_n, 其中$a_i \in \{0, 1\}$
- 给出q个操作, 操作分为两种
- 对于形如1 x的操作, 如果$a_x = 0$, 则将其变为1. 否则变为0
- 对于形如2 x y的操作, 输出下标介于$[x, y]$间的元素中有多少个为1
- $n = 10^6$, $q = 10^6$

解法

- 容易发现，对于操作1，如果$a_x = 1$，那么就是对a_x进行减1操作，否则就是加1操作
- 对于操作2，本质就是输出$\sum_{i=x}^{y} a_i$
- 和上一题一样，树状数组维护即可
- 时间复杂度$O(q \log n)$
- 留作作业

例题10

- 给定一个长度为n的序列$a_1, a_2, \ldots, a_n$
- 给出q个操作，操作分为两种
- 对于形如$1\ x\ y$的操作，将a_x改为y
- 对于形如$2\ x\ y$的操作，输出$\bigoplus_{i=x}^{y} a_i$
- $n = 10^6, q = 10^6$

解法

- 这个问题与之前的问题相比，把求和改成了求异或和
- 使用树状数组求和时，使用了性质$\sum_{i=x}^{y} a_i = \sum_{i=1}^{y} a_i - \sum_{i=1}^{x-1} a_i$
- 因为异或的性质：$a \oplus a = 0$
- 所以可以得到$\oplus_{i=x}^{y} a_i = (\oplus_{i=1}^{y}) \oplus (\oplus_{i=1}^{x-1} a_i)$
- 问题迎刃而解
- 时间复杂度$O(q \log n)$

解法

- 代码:

```
cin >> n >> q;
for(int i = 1; i <= n; i++)
    cin >> a[i];
for(int i = 1; i <= n; i++)
    for(int j = i; j >= i - lowbit(i) + 1; j--)
        c[i] ^= a[j];
```

解法

- while(q--) {

```
    int opt, x, y;
    cin >> opt >> x >> y;
    if(opt == 1) {
        int idx = x;
        int pre = a[x];
        a[x] = y;
        while(idx <= n) {
            c[idx] ^= pre;
            c[idx] ^= y;
            idx += lowbit(idx);
        }
    }
```

解法

- ```
      else if(opt == 2) {
          int idx = y;
          int res = 0;
          while(idx >= 1) {
              res ^= c[idx];
              idx -= lowbit(idx);
          }
          idx = x - 1;
          while(idx >= 1) {
              res ^= c[idx];
              idx -= lowbit(idx);
          }
          cout << res << endl;
      }
  }
  ```
  ```

解法

- 事实上，被操作的代数系统是交换群就可以使用树状数组进行维护
- 拿人话来说就是运算满足交换律、结合律、有逆元即可
- 但是还有一些运算并不满足上述性质，比如max运算
- 如果仅知道$\max_{i=1}^{y} a_i$和$\max_{i=1}^{x-1} a_i$，是无法得到$\max_{i=x}^{y} a_i$的
- 此时使用什么数据结构来维护呢？

例题11

- 给定一个长度为n的序列$a_1, a_2, \ldots, a_n$
- 给出q个操作，操作分为两种
- 对于形如$1\ x\ y$的操作，将a_x改为y
- 对于形如$2\ x\ y$的操作，输出$\max_{i=x}^{y} a_i$
- $n = 10^6$，$q = 10^6$

线段树(Segment Tree)

- 线段树是一种基于分治结构的二叉树
- 为方便理解，先将序列补成2^k长度，其中k满足$2^{k-1} < n, 2^k \geq n$
- 从最底开始，每层两两配对合并至上一层，直至到某一层只剩一个元素
- 容易证明，层数是$O(\log n')$的，总点数为$O(n')$的，其中$n' = 2^k$
- 对某一个元素进行修改时，容易证明每层有且仅有一个元素被影响
- 查询一个区间时，容易证明每层至多有两个元素对答案有直接贡献
- 所以修改操作和查询操作的复杂度都是$O(\log n')$的
- 总复杂度$O(q \log n')$

- 补成2^k长度:

```
cin >> n >> q;
for(int i = 1; i <= n; i++)
    cin >> a[i];
for(l = 1; l < n; l *= 2);
for(int i = n + 1; i <= l; i++)
    a[i] = -INF;
n = l;
```

- 建立线段树:

```
void build(int cur, int l, int r) {
    if(l == r) st[cur] = a[l];
    else {
        int mid = (l + r) / 2;
        build(cur * 2, l, mid);
        build(cur * 2 + 1, mid + 1, r);
        st[cur] = max(st[cur * 2], st[cur * 2 + 1]);
    }
}
```

- 修改元素:

```
void modify(int cur, int l, int r, int pos, int val) {
    if(l == r) st[cur] = val;
    else {
        int mid = (l + r) / 2;
        if(pos <= mid) modify(cur * 2, l, mid, pos, val);
        else modify(cur * 2 + 1, mid + 1, r, pos, val);
        st[cur] = max(st[cur * 2], st[cur * 2 + 1]);
    }
}
```

- 查询区间max:

```
int query(int cur, int l, int r, int x, int y) {
    if(x <= l && y >= r) return st[cur];
    int mid = (l + r) / 2, res = -INF;
    if(y <= mid) res = query(cur * 2, l, mid, x, y);
    else if(x > mid)
        res = query(cur * 2 + 1, mid + 1, r, x, y);
    else {
        int ls = query(cur * 2, l, mid, x, y);
        int rs = query(cur * 2 + 1, mid + 1, r, x, y);
        res = max(ls, rs);
    }
    return res;
}
```

解法

- 本题:

```
while(q--) {
    int opt, x, y;
    cin >> opt >> x >> y;
    if(opt == 1) modify(1, 1, n, x, y);
    else printf("%d\n", query(1, 1, n, x, y);
}
```

线段树(Segment Tree)

- 实际上，即使不补成2^k长度这样做也是正确的
- 但是如果不补齐的话，就不能看成从下向上合并了
- 两者本质是一样的，只是理解起来不同
- 大家可以思考一下为什么可以不补齐

例题11

- 给定一个长度为n的序列a_1, a_2, ..., a_n
- 给出q个操作，操作分为两种
- 对于形如1 x y的操作，将a_x改为y
- 对于形如2 x y的操作，询问$[x, y]$这段区间的值是否单调不减，如果是输出"YES"，否则输出"NO"
- $n = 10^6$, $q = 10^6$

解法

- 线段树维护的信息只要支持合并即可
- 除了运算结果外，还可以维护各种各样的信息
- 比如本题，线段树上每个节点维护其表示的区间的最左端的数字、最右端的数字、该区间是否为不减区间，分别设为ln, rn, $flag$
- 合并信息的时候，显然$ln_{cur} = ln_{cur\times 2}$，$rn_{cur} = rn_{cur\times 2+1}$，如果$rn_{cur\times 2} \le ln_{cur\times 2+1}$，那么$flag_{cur} = \max(flag_{cur\times 2}, flag_{cur\times 2+1})$，否则$flag_{cur} = 0$
- 查询时，将有直接贡献的$O(\log n)$个点的信息如上合并，得到答案
- 时间复杂度$O(q\log n)$
- 留作作业

例题12

- 给定一个长度为n的序列a_1, a_2, ..., a_n
- 初始时有$\forall i \in [1, n], a_i = i$
- 给出q个操作，操作分为两种
- 对于形如1 x y的操作，将序列中所有等于a_x的元素的值改为a_y
- 对于形如2 x y的操作，如果$a_x = a_y$，输出"YES"，否则输出"NO"
- $n = 10^6$, $q = 10^6$

解法

- 直接模拟的复杂度为$O(qn)$, 不能通过本题
- 按值分类, 每一种值的元素属于且仅属于一个集合
- 则初始时有n个集合, 每个集合中有1个元素
- 修改操作本质上就是将两个集合合并
- 查询操作本质上就是判断两个元素是否属于一个集合

并查集(Union-Find Sets)

- 维护这些集合，可以使用一种叫做并查集的数据结构
- 定义一个序列$\{fa_n\}$，fa_i表示值为i的元素所属的集合
- 初始时有$\forall i \in [1, n], fa_i = i$
- 查询x所属的集合时，如果$fa_x = x$，答案就是x，否则令$x = fa_x$，重复上述操作直至得到答案
- 合并x, y所属的集合时，直接令$fa_{find(x)} = fa_{find(y)}$即可
- 时间复杂度$O(nq)$

并查集(Union-Find Sets)

- 代码:

```
int find(int cur) {
    if(fa[cur] == cur)
        return cur;
    return find(fa[cur]);
}
void merge(int x, int y) {
    if(find(x) == find(y))
        return;
    fa[find(x)] = find(y);
}
```

并查集(Union-Find Sets)

- 这样的复杂度显然不能满足要求，但是只需要将find函数改为如下

```
int find(int cur) {
    if(fa[cur] == cur)
        return cur;
    return fa[cur] = find(fa[cur]);
}
```

- 时间复杂度即可成为$O(n+q\alpha(n))$，其中$\alpha(n)$为Ackerman函数的某个反函数，在很大的范围内这个函数的值可以看成是不大于4的，所以并查集的操作可以看作是线性的
- 为什么会这样?
- 可见Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 21: Data structures for Disjoint Sets, pp. 498 – 524.

解法

- 本题:

```
cin >> n >> q;
for(int i = 1; i <= n; i++)
    fa[i] = i;
while(q--) {
    int opt, x, y;
    cin >> opt >> x >> y;
    if(opt == 1) merge(x, y);
    else {
        if(find(x) == find(y))
            cout << "YES" << endl;
        else
            cout << "NO" << endl;
    }
}
```

接下来是感人至深的习题

- 反正也没有几题，讲完就下课
- 题解课件上毛都没有哈哈哈哈

一个习题

- 给定一个长度为n的序列a_1, a_2, ..., a_n
- 给出q个操作，操作分为两种
- 对于形如1 x y z的操作，将序列中下标介于$[x, y]$的元素加上z
- 对于形如2 x y的操作，输出$\sum_{i=x}^{y} a_i$
- $n = 10^5$, $q = 10^5$

一个习题

- 给定一个长度为n的序列$a_1, a_2, ..., a_n$
- 给出q个操作，操作分为两种
- 对于形如1 x y的操作，将a_x改为y
- 对于形如2 x y的操作，判断下标介于$[x, y]$的元素能能否构成一个等差数列，如果能，则输出"YES"，否则输出"NO"
- $n = 10^5$, $q = 10^5$

一个习题

- 给定一个长度为n的序列$a_1, a_2, ..., a_n$
- 给出q个操作，每个操作形如$x\ y$
- 对于每个操作，输出下标介于$[x, y]$的元素中共有多少种不同的值
- $n = 10^5, q = 10^5$

结束

Thanks!