Block Decomposition (Part - 01) - MO's Algo

One Trick to rule 'em all

Posted on September 21, 2017

MO's Algo জিনিসটা দিন দিন জনপ্রিয় হয়ে উঠছে। দেখা যাক এই MO's Algo কি।

MO's Algo আসলে একটা Offline Query Ordering Trick, এর সাহায্যে কোন N Size এর Array / Tree বা অন্যকিছুর উপরে Q টা Range / Path Query করা যায় কিছু শর্ত সাপেক্ষে। মোট Complexity হয় $O((N+Q)\sqrt{N} \times P(n))$। আর যদি Update থাকে তাহলেও করা যায় $O(Q \times N^{\frac{2}{3}})$ তে।

তবে এইটা Offline Trick, মানে আমাদের আগে থেকে সমস্ত Query + Update জানা থাকা লাগবে, এর পরে আমরা Query গুলা উলটা পালটা করে সমাধান করব এর পরে আবার যেই Order এ Query দেওয়া হয়েছিল সেইভাবে Print করব। 🙂

[**Note**: যারা MO's Algo পারেন কিন্তু Update পারেন না তারা MO's Algo Basic Idea Skip করতে পারেন। তবে MO's Algo এর Complexity Analysis টা অবশ্যই ভাল করে জানতে হবে MO with Update বুঝতে চাইলে। MO's Algo তে নতুন হলে আমি বলব with Update Part তা আপাতত বাদ দিতে।]

Basic Idea

Basic Idea টা বুঝার জন্য আমরা একটা সহজ প্রবলেম নেই। ধরি Range Sum Query টাই সল্ভ করতে চাই একটু অন্য ভাবে।

> "একটা Array দেওয়া আছে, N size এর। আর Q টা Query আছে - Array এর $[l, r]$ Range এর Index গুলাতে যেসব সংখ্যা আছে তাদের যোগফল কত?

First Try

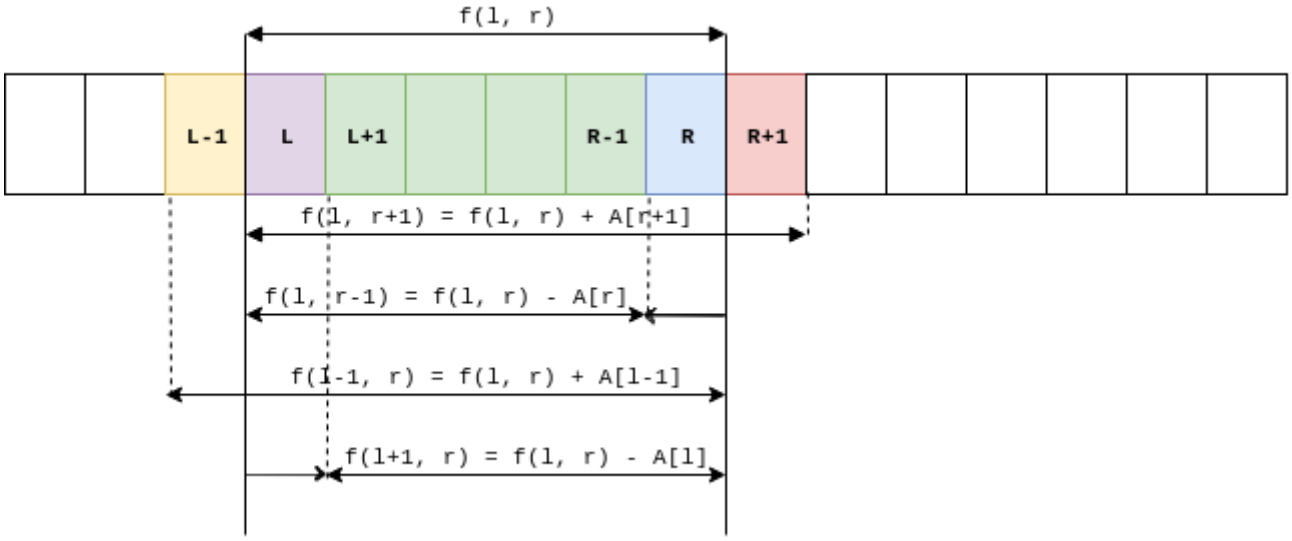
ধরি আমাদের এইখানে - $f(l, r) = \sum_{i=l}^{r} A_i$

এখন লক্ষ্য করি, আমরা যদি $f(l, r)$ এর মান জানি তাহলে কি $f(l+1, r), f(l-1, r), f(l, r+1), f(l, r-1)$ এই গুলা বের করা কি খুব সহজ না? $f(l, r+1)$ এর মধ্যে $f(l, r)$ থেকে কি কম বা বেশি আছে?

বেশি কিছু পার্থক্য নাই! $f(l, r+1)$ এর মধ্যে খালি A_{r+1} টা বেশি আছে। তাই $f(l, r+1) = f(l, r) + A_{r+1}$। একই ভাবে -

$$f(l, r+1) = f(l, r) + A_{r+1}$$
$$f(l, r-1) = f(l, r) - A_r$$
$$f(l-1, r) = f(l, r) + A_{l-1}$$
$$f(l+1, r) = f(l, r) - A_l$$

নিচের ছবিটা দেখলে আশাকরি বুঝা যাবে -



তাহলে আমরা এইভাবে Query গুলা Process করতে পারি -

- শুরুতে আমরা ২টা Pointer নেব - $L = 0$ আর $R = -1$, আর একটা Variable sum, যেইটা $[L, R]$ Range এর সংখ্যা গুলার যোগফল store করবে।
- ২টা Function Define করি -
- `add(x)` যেইটা Sum Variable এ A_x এর Contribution Add করে।
- `remove(x)` এটা Sum Variable এ A_x এর Contribution Remove করে।
- এখন একটা Query নেব - $[l, r]$, আমাদের লক্ষ্য হল $[L, R]$ range এর L আর R কমিয়ে / বাড়িয়ে l, r এর সমান করা। তাহলেই sum Variable এ আমাদের উত্তর পেয়ে যাব!!

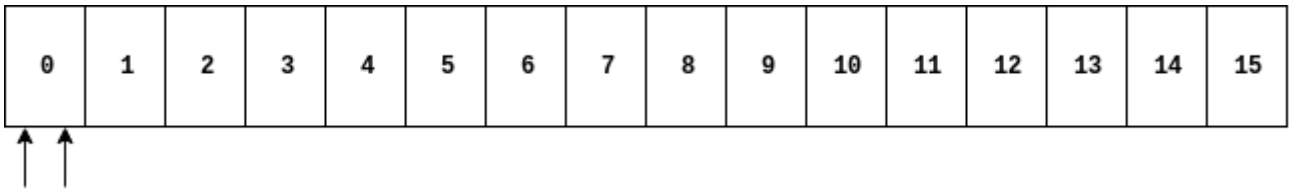
Code হতে পারে এরকম -

```
struct query{
  int l, r, id;
} q[maxn];

int l = 0, r = -1, sum = 0, ans[maxn];

void add(int x) { sum += a[x]; }
void remove(int x) { sum -= a[x]; }

int main() {
  // do stuff, take input etc...
  for(int i = 0; i < Q; i++) {
    cin >> q[i].l >> q[i].r;
    q[i].id = i;
  }
  for(int i = 0; i < Q; i++) {
    while(l > q[i].l) add(--l);
    while(r < q[i].r) add(++r);
    while(l < q[i].l) remove(l++);
    while(r > q[i].r) remove(r--);
    ans[q[i].id] = sum;
  }
}
```

তাহলে এখন `ans[i]` তে i^{th} Query এর answer পাওয়া যাবে।

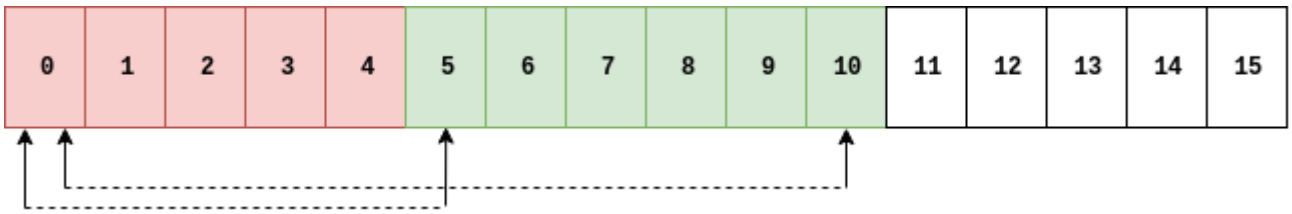
একটা উদাহরণ দেখা যাক - আমাদের প্রথম Query হল $[5, 10]$.

আমাদের শুরুর Left pointer আর Right pointer আছে $[0, 0]$ তে।

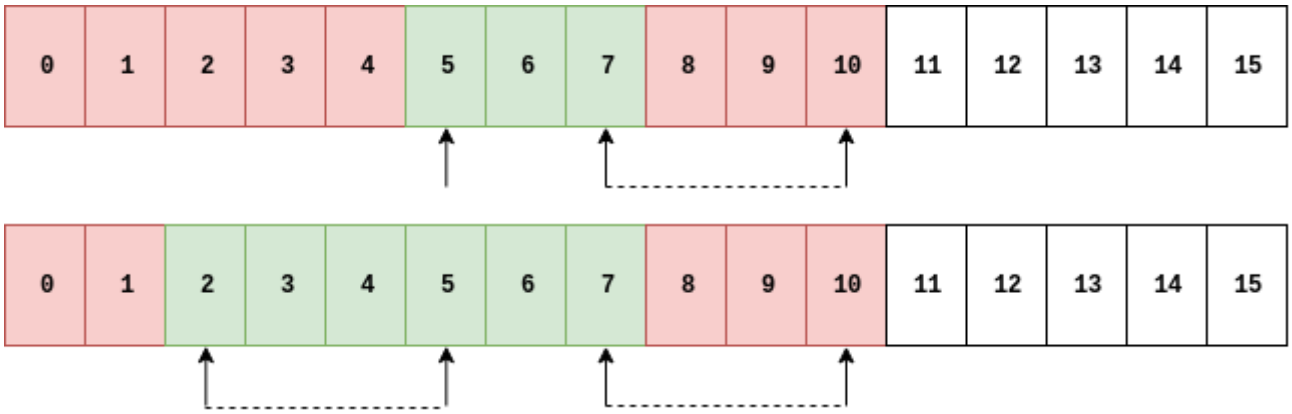


আমরা Right pointer কে টেনে Index 10 এ নেব। আর মাঝখানের সব গুলা Index এর যোগফল sum এ Add করব।

আবার Left Pointer কে 0 থেকে 5 এ আনব। আর মাঝের সব গুলা কে Remove করে দেবও sum থেকে।



এর পরের Query মনে করি $[2, 7]$ তাহলে এইরকম হবে যে, Right Pointer 7 থেকে 5 এ যাবে আর মাঝখানের সব Remove হবে। এর পরে Left Pointer 5 থেকে 2 তে যাবে আর মাঝের সব Add হবে -



কিন্তু আমাদের এই Approach বেশি Efficient না। যেমন নিচের Query গুলা দেখা যাক, $N = 100000$ ধরে নেই -

- $[100000, 100000]$
- $[1, 100]$
- $[100000, 100000]$
- $[1, 10]$

প্রথম Query এর জন্য Left Pointer 0 থেকে 100000 পর্যন্ত যায়, তেমনি Right Pointer ও 0 থেকে 100000 পর্যন্ত যায়।

কিন্তু এর পরের Query তেই আবার Left Pointer কে 100000 ঘর আগে আসতে হয় আবার Right Pointer কেও অনেক ঘর পার করতে হয়। এভাবে করলে Worst Case এ আমাদের প্রত্যেক Query এর জন্য $O(n)$ লেগে যাবে।

তাহলে মোট Complexity $O(QN)$ যেইটা অনেক খারাপ বলা যায়।

Observation

কেবল যেই Approach টা দেখলাম সেইটার একটা বিশেষত্ব আছে। এই Approach খালি Query গুলা আমরা কোন Order এ করছি তার উপরে নির্ভর করে। পাশাপাশি ২টা Query এর l, r এর পার্থক্য যদি কম হয় তাহলে আমদের Algo অনেক ভাল কাজ করবে। কিন্তু Input এ তো এইভাবে থাকবে তার কোন Surety নাই। তাহলে আমরা কি Query গুলাকে কোন ভাবে Sort করে নিতে পারি?

চেষ্টা করি, আমরা Query গুলাকে Left Side এর উপরে ভিত্তি করে যদি Sort করি তাহলে কি হয়?

তাহলে সব গুলা Query মিলিয়ে Left Side মাত্র $O(n)$ বার Move করবে।

কিন্তু, ব্যাপার হল, Left Side এর উপরে ভিত্তি করে Sort করলে Left Side কম Move করে ঠিকই, কিন্তু Right side অনেক বেশি সরে যেতে পারে।

যেমনঃ আমাদের Query গুলা যদি এমন হয় -

- $[1, 1]$
- $[2, 100000]$
- $[3, 10]$
- $[4, 100000]$
- $[5, 20]$
- $[6, 100000]$
- $[7, 30]$
- $[8, 100000]$
- $[9, 40]$
- $[10, 100000]$

তাহলে আমাদের Left Pointer খুবি কম Move করবে। সব Query মিলালে মোট $O(n)$ বার। কিন্তু Right Pointer প্রত্যেক Query তে $O(n)$ Move করতে পারে। আবার সেই $O(QN)$ Complexity। 😢

আবার চেষ্টা করি, যদি Right Side এর উপরে ভিত্তি করে Sort করি? তাহলেও আমাদের Right Pointer সব Query মিলালে মাত্র $O(n)$ বার Move করবে। কিন্তু Left Pointer অনেক বেশি Move করতে পারে।

তাহলে আর কিভাবে Sort করব? এইখানেই আসে MO's Ordering!

MO's Ordering আর Complexity Analysis সম্পুর্ণ না বুঝা পর্যন্ত বারবার পড়ার অবুরধ রইল

MO's Ordering

MO's Ordering এর মাধ্যমে এইসব সমস্যা দুর করা যায়। MO's Order এ সমস্ত Query কে l বা r এর উপরে Based করে Sort করা হয় না।

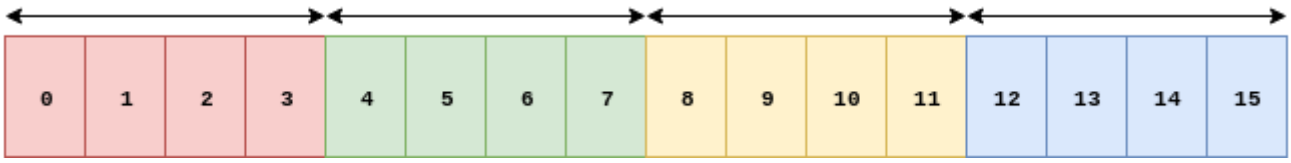
এক্ষেত্রে Array কে k-size এর Block এ ভাগ করা হয়। তাহলে এইরকম Block থাকবে $\left\lceil \frac{n}{k} \right\rceil$ টা। শেষের Block এর Size কিছু কম থাকতে পারে।

মানে Block 0 এ থাকবে $[0, k-1]$ Range এ যাদের Left side আছে। Block 1 তে থাকবে $[k, 2k-1]$ range এর Query গুলা ইত্যাদি। মানে একটা l Left Side আছে এমন Query থাকবে $\left\lfloor \frac{l}{k} \right\rfloor$ নাম্বার Block এ।

এবার আমরা একটা করে Block Process করব। একই Block এ যেই Query গুলা আছে তাদের আমরা Right Side এর ভিত্তিতে Sort করব।

এত ঝামেলা কর লাভ কি হল? দেখা যাক -

ধরি, $n = 16$ আর $k = 4$। তাহলে আমাদের Block এ ভাগ করার পরে Array হবে -



এখন ধরি Query গুলা হল -

- $[0, 15]$
- $[1, 4]$
- $[2, 15]$
- $[3, 5]$
- $[4, 15]$
- $[5, 6]$
- $[7, 15]$
- $[6, 6]$

সরাসরি এদের এইভাবে Process করতে গেলে বুঝাই যাচ্ছে যে প্রত্যেক Query তে Right Pointer $O(n)$ বার Move করে। কিন্তু এইবার আমরা এদের Block এ ভাগ করে বিন্যাস করি। এর পরে একই Block এর Query গুলাকে Right Side অনুযায়ী Sort করি -

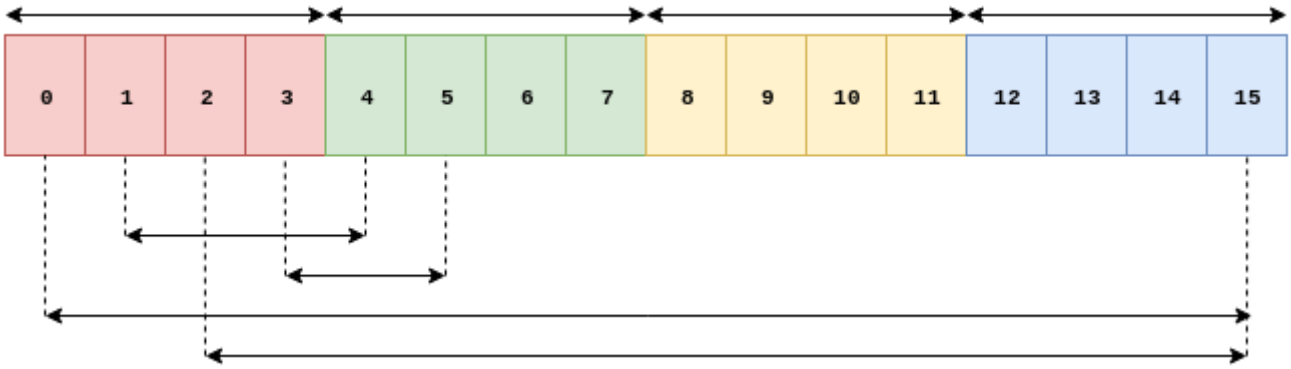
Block-0

- $[1, 4]$
- $[3, 5]$
- $[0, 15]$
- $[2, 15]$

Block-1

- $[5, 6]$
- $[6, 6]$
- $[4, 15]$
- $[7, 15]$

বুঝতে সমস্যা হলে নিজে একবার এঁকে দেখার অনুরোধ রইল। যেমন প্রথম Block এর Query গুলা এইরকম আছে -



এখন দেখা যাক এতে আমাদের কি উন্নতি হল। আমরা যদি আগের মত করে Query Process করি তাহলে প্রথম 4 টা Query তে আমাদের Left pointer + Right pointer কয়বার Move করে দেখা যাক।

- $[0, 15]$
- $[1, 4]$
- $[2, 15]$
- $[3, 5]$

শুরুতে $|0 - 0| + |0 - 15| = 15$, এর পরের Query তে $|15 - 4| + |1 - 0| = 12$। এর পরে $|4 - 15| + |1 - 2| = 12$, শেষে $|15 - 5| + |2 - 3| = 11$

মোট $15 + 12 + 12 + 11 = 50$

কিন্তু MO's Ordering এর পরে -

- $[1, 4]$

- $[3, 5]$
- $[0, 15]$
- $[2, 15]$

আবার হিসাব করি -

$$
\begin{aligned}
&|0-1|+|0-4|+\\
&|1-3|+|4-5|+\\
&|3-0|+|5-15|+\\
&|0-2|+|15-15|\\
&=(1+2+3+2)+(4+1+10+0)\\
&=8+15=23
\end{aligned}
$$

কোথায় 23 আর কোথায় 50। 😕

কিন্তু এত কম লাগল কেন? কারণ একটা Block এর সব Query এর জন্য আমাদের Right pointer মাত্র একবার 0 থেকে 15 তে গেছে। তাই Right pointer এর Total movement = 15, বা $O(n)$। আর 2 টা Query এর Left Side এর পার্থক্য একটা Block এ সর্বোচ্চ $O(k)$. উপরের উদাহরণ এ ২টা পাশাপাশি Query এর জন্য তাই Left pointer সর্বোচ্চ 4 বার Move করছে।

Complexity Analysis

তাহলে মোট Complexity বের করা যাক।

আমাদের Complexity হবে $O(\text{Left Pointer Move} + \text{Right Pointer Move})$

আমাদের k-size এর $\frac{n}{k}$ টা Block আছে।
একটা Block এর মধ্যে Right pointer সর্বমোট $O(n)$ বার Move করে। তাহলে $\frac{n}{k}$ টা Block এর জন্য $O(n \times \frac{n}{k}) = O(\frac{n^2}{k})$ বার।

একটা Block এর মধ্যে Left pointer একটা Query এর জন্য সর্বোচ্চ $O(k)$ বার move করে। তাহলে মোট $O(qk)$, q টা Query এর জন্য।

Total Complexity $O(\frac{n^2}{k} + qk)$

এখন আমাদের এমন একটা k বের করতে হবে যেন এইটার মান সবচেয়ে ছোট হয়। হিসাবের সুবিধার্থে ধরে নেই $q = O(n)$

লক্ষ্য করি, k বাড়লে $\frac{n^2}{k}$ কমে, কিন্তু nk বাড়ে।
আবার k কমলে $\frac{n^2}{k}$ বাড়ে, nk কমে।

তাহলে $\frac{n^2}{k} + nk$ এর মান সর্বনিম্ন হবে যদি $\frac{n^2}{k} = nk$ হয়। এখন -

$$\begin{aligned} \frac{n^2}{k} &= nk \\ \implies n^2 &= nk^2 \\ \implies k^2 &= n \\ \implies k &= \sqrt{n} \end{aligned}$$

তার মানে $k = O(\sqrt{n})$ নিলে আমরা সবচেয়ে ভাল Complexity পাব। তাহলে মোট Complexity = $O(\frac{n^2}{\sqrt{n}} + q\sqrt{n}) = O(n\sqrt{n} + q\sqrt{n}) = O((n+q)\sqrt{n})$।

এখন আমাদের `add(x)` আর `remove(x)` এর Complexity যদি $O(P(n))$ হয় তাহলে মোট Complexity $O((N+Q)\sqrt{N}P(N))$

Note: আমরা ধরে নিছিলাম $Q = O(n)$। তবে কিছু Problem এ এরকম নাও হতে পারে। N অনেক বড়, Q ছোট অথবা Q বড় কিন্তু N ছোট থাকতে পারে। সেক্ষেত্রেও $k = \sqrt{n}$ নিলে সমস্যা হইয়ার কথা না। কিন্তু সমস্যা হলে $k = \sqrt{\frac{n^2}{Q}}$ নিয়ে চেষ্টা করা যেতে পারে। মূলত k এর এই মানের জন্যই MO's Ordering Best Performance দেয়। 🙂

Sample Implementation

একটু আগে যেই Code টা দেওয়া হল সেইটাতেই মাত্র আর ২টা Line Add করলেই হয়ে যাবে। একটা Compare Function লিখতে হবে যেইটা ২টা Query এর মধ্যে কে আগে হবে সেইটা বলে দেবে। যদি ২টা আলাদা Block এ হয় তাহলে যে আগে সে আগে হবে। আর ২টা একই Block এর বলে যার Right Side আগে আছে সে আগে হবে। এর পরে Query Process শুরু করার আগে Sort করে নিলেই হল। দেখা যাক -

```
struct query{
  int l, r, id;
} q[maxn];

const int k = 320; // As sqrt(100000) = ~320
                   // I recommand setting the max block size
                   // for the problem at the beginning.
                   // Somehow it fastens up runtime.

bool cmp(query &a, query &b) {
  int block_a = a.l / k, block_b = b.l / k;
  if(block_a == block_b) return a.r < b.r;
  return block_a < block_b;
}

int l = 0, r = -1, sum = 0, ans[maxn];

void add(int x) { sum += a[x]; }
void remove(int x) { sum -= a[x]; }

int main() {
  // do stuff, take input etc...
  for(int i = 0; i < Q; i++) {
    cin >> q[i].l >> q[i].r;
    q[i].id = i;
  }
  sort(q, q+Q, cmp);
  for(int i = 0; i < Q; i++) {
    while(l > q[i].l) add(--l);
    while(r < q[i].r) add(++r);
    while(l < q[i].l) remove(l++);
    while(r > q[i].r) remove(r--);
    ans[q[i].id] = sum;
  }
}
```

MO's Ordering কত Powerful!! খালি একটা Ordering করেই $O(QN)$ এর Solution কে $O(Q\sqrt{N})$ করে ফেলল!! 😉

Problem Solving

MO's Algo এর Problem গুলা সাধারণত Codeforces এ Div1C/D তে থাকে। কিন্তু MO' Algo জানলে Solution হয় খুবই সোজা। কিছু প্রবলেম দেখা যাক -

Problem 1:

DQUERY (https://www.spoj.com/problems/DQUERY/): সংক্ষেপে প্রবলেম টা হল -

> n size এর একটা Array আছে। Q টা Range query আছে। একটা $[l, r]$ range এর query এর উত্তর হল $A[l \cdots r]$ range এ কয়টা Distinct Number আছে।

Hint: Distinct Number Count করার একটা উপায় হল যেই সব Number এর Count > 1 আছে তাদের একবার Count করা।

তাহলে আমাদের `add(x)` আর `remove(x)` এর জন্য একটা Count array লাগবে। `add(x)` function এ আমরা একটা Number এর Count বাড়াব। Count যদি 1 হয়ে যায় তাহলে আমাদের Ans কে Increase করব, তাহলে 1 থেকে যখন বড় হবে তখন আর সেইগুলাকে Count করা হবে না। আর `remove(x)` এ Count কে কমাতে হবে। এইটা 0 হয়ে গেলে Ans কে Decrease করতে হবে। তাহলে Function ২টা এরকম হতে পারে -

```
long long cnt[1000100], ans = 0;
void add(int x) {
  x = a[x];
  cnt[x]++;
  if(cnt[x] == 1) ans++;
}
void remove(int x) {
  x = a[x];
  cnt[x]--;
  if(cnt[x] == 0) ans--;
}
```

Problem 2:

CF 86D (http://codeforces.com/problemset/problem/86/D) (Div1D 😱): সংক্ষেপে প্রবলেম টা হল -

> n size এর একটা Array আছে। Q টা Range query আছে। একটা $[l, r]$ range এর query এর উত্তর হল প্রত্যেক $\sum_{x=1}^{\infty} C_x \times C_x \times x$. এখানে $C_x = x$, $A[l \cdots r]$ range এর মধ্যে কয়বার আসছে।
> যেমনঃ $[l, r]$ range এর সংখ্যা গুলা $[1, 1, 3, 2, 3, 2, 2, 1]$ হলে উত্তর হবে
> $C_1 \times C_1 \times 1 + C_2 \times C_2 \times 2 + C_3 \times C_3 \times 3 + \cdots$
> $= (3 \times 3 \times 1) + (3 \times 3 \times 2) + (2 \times 2 \times 3) = 9 + 18 + 12 = 39$

Problem টা দেখে অনেক কঠিন মনে হলেও MO's Algo দিয়ে অনেক সহজে Solve করা যাবে। আমাদের খালি `add(x)` আর `remove(x)` function ২টা পরিবর্তন করতে হবে।

এই প্রবলেম এ আমাদের `add(x)` এ তাহলে কি করতে হবে? একটা সংখ্যা বেড়ে গেলে কিন্তু শুধু ওই সংখ্যা টা যোগ করলেই হচ্ছে না। ওই সংখ্যার Count ও বেড়ে যাচ্ছে, এর পরে নতুন Count এর বর্গ গুন ওই সংখ্যা Sum এ Contribute করছে। তাহলে আমাদের আগে পুরানো Count এর Contribution remove করতে হবে। Count বাড়াতে হবে। আবার নতুন Contribution add করতে হবে। আর `remove(x)` এও একই কাজ করব, কিন্তু Count কমাতে হবে। তাহলে Function টা হতে পারে এইরকম -

```
long long cnt[1000100], sum = 0;
void add(int x) {
  sum -= cnt[a[x]] * cnt[a[x]] * a[x];
  cnt[a[x]]++;
  sum += cnt[a[x]] * cnt[a[x]] * a[x];
}
void remove(int x) {
  sum -= cnt[a[x]] * cnt[a[x]] * a[x];
  cnt[a[x]]--;
  sum += cnt[a[x]] * cnt[a[x]] * a[x];
}
```

Problem 3:

Sherlock and Inversions (https://www.hackerearth.com/practice/data-structures/advanced-data-structures/fenwick-binary-indexed-trees/practice-problems/algorithm/sherlock-and-inversions/): সংক্ষেপে প্রবলেম টা হল:

> n size এর একটা Array আছে। Q টা Range query আছে। একটা $[l, r]$ range এর query এর উত্তর হল কয়টা pair (i, j) আছে যেন $l \leq i \leq j \leq r$ এবং $a_i > a_j$। মানে Inversion আরকি।

এইটার Solution একটু কঠিন লাগতে পারে। Inversion Problem সম্পর্কে খুব ভাল ধারনা থাকা লাগবে।

এই প্রবলেম এ Left side এর Add আর Right side এর Add আলাদা।
যখন Right side এ একটা নতুন Element range এ Add করবও তখন এই নতুন Element টা কয়টা Inversion Create করে সেইটা Add করতে হবে।

Right Side এ একটা Element বেশি থাকলে সেইটা কয়টা Inversion Create করে? আগের $[l, r]$ range এ কয়টা Elemnet A_{r+1} থেকে বড় আছে! এইটা কে current inversion এর সাথে যোগ করতে হবে। এইটা আমরা একটা BIT দিয়ে করতে পারি।

একই ভাবে বাকি Case গুলা করা যাবে। Left side সামনে বা পিছনে করলে কি হয়, Right side সামনে পিছনে করলে কি হয়।
Inversion Problem সম্পর্কে ভাল ধারনা থাকলে আশাকরি এইটা Solve করতে পারবে সবাই 🙂

Problem 4:

CF 220B (http://codeforces.com/problemset/problem/220/B): এইটা তুলনামূলক সহজ Problem, উপরের গুলা Solve করে থাকে এইটা সবাই পারবে আসা করি।

Problem 5:

Substring Count (https://www.hackerearth.com/problem/algorithm/substrings-count-3/): সংক্ষেপে প্রবলেমটা হল -

> একটা n Length এর Array of string A দেওয়া আছে। Query গুলা হল Array এর $[l, r]$ index range এ কয়টা String এর মধ্যে আরেকটা Given String, Substring আকারে আছে।

এইটা অনেক ভাব সমাধান করা যায়। একটা উপায় হল Hashing করা, Hash গুলার Count কমান/বাড়ান। এর পরে Answer বের করার সময় ওই Hash কয়বার আসছে দেখা। তবে ভাল Implementation না হলে TLE হবে।

আরেকটা উপায় হল `add/remove` function এ কোন String Related DS এ Insert/Delete করা যাতে একটা String কয়বার Substring হিসাবে আসছে সেইটা তাড়াতাড়ি বের করা যায়।

MO’s Algo With Updates

CAUTION: PRO STUFF. Solve more MO’s algo problem before trying to understand this. Also read the MO’s algo complexity part again if you have some gap.

একটা সাধারণ প্রবলেম দেখা যাক -

> একটা n size এর Array A আছে। Q টা Query আছে। হয় আমাদের বের করতে হবে $[l, r]$ index range এ কয়টা Distinct number আছে, নাহয় $A_x = y$ set করতে হবে।

MO’s Algo টা তো Offline এ সব Query solve করে। কিন্তু আমাদের যদি প্রবলেম এ Update থাকবে বলা থাকে তাহলে তো আমরা Query গুলাকে উলটা পালটা করলে Answer পরিবর্তন হতে যাবে। তাহলে Update সহ আবার Offline করা যায় কি করে?

এর জন্য আরেক্টু Smart Way তে Offline Solve করতে হবে। আমরা Query আর Update গুলাকে আলাদা করে ফেলব, মানে ভিন্ন Array তে Store করব।

আমরা Query কে Represent করব ৪টা জিনিস দিয়ে।

```
struct query {
  int l, r, t, id;
} q[maxn];
```

এইখানে t হল এই Query এর আগে কয়টা Update হয়েছে। আবার Update কেও Represent করতে হবে ৩টা সংখ্যা দিয়ে -

```
struct update {
  int x, pre, now;
} u[maxn];
```

এখানে pre হল A_x আগে কি ছিল সেইটা, আর now হল নতুন কি দিয়ে Update করলাম।

কিন্তু এইটা Preprocess করতে একটু ঝামেলা আছে। যেমন পর পর যদি ২টা Same Index এ Update হয়? তাহলে তো আগের Update এর পরে যেইটা হয়েছিল সেইটা Update করতে হবে। এইটা Preprocess করা যেতে পারে এইভাবে -

```
int last[N];
for(int i = 0; i < N; i++)
  last[i] = a[i];

for(int i = 0; i < Q; i++) {
  if( this is a query ) {
    store query {l, r, idx, id++} // idx is number of updates before, id is tl
  }
  if( this is an update ) {
    u[++idx] = {x, last[x], y};
    last[x] = y;
  }
}
```

এইভাবে করলে আমাদের Main Array অপরিবর্তিত থাকল।

এখন আমাদের আরেকটা function লাগবে `add(x)/remove(x)` এর মতই, ধরি `apply(x, y)` , এইটা A_x কে y দিয়ে Change করলে কি হয় সেইটা Note করবে। যেমন এই প্রবলেম এর জন্য এইটা এমন হতে পারে -

```
void apply(int x, int t) {
  if(l <= x <= r) {  // l, r is the l, r from MO's algo
    remove(x);
    a[x] = y;
    add(x);
  } else a[x] = y;
}
```

মানে আগে আমরা বর্তমান A_x এর Contribution Remove করে দিলাম, নতুন সংখ্যা বসালাম, আবার নতুন Number এর Contribution add করলাম। তবে একটা জিনিস লক্ষ্য করি, যদি Index x, আমাদের বর্তমান এ MO's algo এর যেই Left side আর Right side আছে তার মধ্যে যদি না থাকে তাহলে নিশ্চয়ই তাদের কোন Contribution ও নাই। এজন্য আমরা খালি ওই Index মূল Array তে Update করে দিলেই হবে।

এখন আমরা MO's Algo চালাতে পারি। আমাদের Left Pointer, Right Pointer এর সাথে একটা Time Pointer ও লাগবে। এইটা হিসাব রাখবে বর্তমানে কয়টা Update দেওয়া হয়ে গেছে Array তে।

Update Represent করার সময় আগের Value রাখার সুবিধা কি? এখন আমরা চাইলে Update Reverse করতে পারি। একটা Update Reverse মানে Just `u[i].pre` দিয়ে Update করা!

তাহলে আমরা একটা Query Process করার সময় ঠিক যত গুলা Update তার আগে হয়েছে Note করে রেখেছি সেই গুলা রেখে বাকি গুলা Remove করে দেব। অথবা অতগুলা Update না হয়ে থাকলে সেইগুলা Apply করব।

তাহলে Query Solve করার জায়গাটা হবে এমন -

```
int l = 0, r = -1, t = 0; // Note that these values may change
                          // Depending on your implementation of other part
for(int i = 0; i < Q; i++) {
  while(t < q[i].t) t++, apply(u[t].x, u[t].now); // Forward Update
  while(t > q[i].t) apply(u[t].x, u[y].pre); // Reverse Update

  while(l > q[i].l) add(--l);
  while(r < q[i].r) add(++r);
  while(l < q[i].l) remove(l++);
  while(r > q[i].r) remove(r--);

  ans[q[i].id] = some_variable;
}
```

কিন্তু!! এই Approach $O(QN)$ Bruteforce থেকেও খারাপ!
যদি আমরা MO's Ordering ব্যবহার করি তাহলে কি হয় দেখা যাক -

ধরি, $N = 100000$ আর Query গুলা এইরকম -

1. $[1, 1]$
2. $[1, 3]$
3. $[1, 5]$
4. $[1, 7]$
5. $\cdots$

$\cdots\cdots$

25000. Update
25001. Update

25002. Update
25003. Update
25004. $\cdots$
$\cdots\cdots$
75000. $[1, 2]$
75001. $[1, 4]$
75002. $[1, 6]$
75003. $[1, 8]$
$\cdots\cdots$

এখন আমাদের আগের মত করে যদি শুরুতে Left Side কোন Block এ পরে, এর পরে কার Right Side আগে এর ভিত্তিতে Sort করি, তাহলে এইরকম হবে -

1. $[1, 1]$
75000. $[1, 2]$
2. $[1, 3]$
75001. $[1, 4]$
3. $[1, 5]$
75002. $[1, 6]$
4. $[1, 7]$
75003. $[1, 8]$
5. $\cdots$
75004. $\cdots$
$\cdots\cdots$

এইখানে সব Query একই Block এ আছে। শুধু Query থাকলে আমাদের খালি Right Pointer $O(n)$ Move করত। এখনো একটা Block এ Right Pointer $O(n)$ Move করছে।
কিন্তু এখন পাশাপশি ২টা Query এর মধ্যে যত Update আছে সব পরে গেছে। মানে $O(n)$ টা Update। একটা থেকে আরেকটাতে যেতে গেলে মাঝখানে 50000 ঘর Time Pointer কে Move করতে হচ্ছে!!! প্রত্যেক বার Time Pointer $O(n)$ move করতে পারে। তাহলে মোট Complexity $O(\text{Left} + \text{Right} + \text{Time Pointer move}) = O(n\sqrt{n} + q\sqrt{n} + qn)$, **যেইটা Bruteforce $O(QN)$ থেকেও খারাপ।** 😂

তার মানে আমাদের এইভাবে Query Sort করা যাবে না। এমন ভাবে করতে হবে যেন Left Pointer, Right Pointer, Time pointer সবাই কম বার Move করে।

MO Without Update করার সময় আমরা শুরুতে একবার L based sort করার try করছিলাম। তখন দেখা যায় প্রত্যেক Query তে Right Pointer $O(n)$ move করে। আবার এখন আমাদের কি হচ্ছে? আমরা Left Block আর Right Side দিয়ে Sort করলে Time pointer $O(n)$ Move করতেছে। তাহলে আমাদের Target হল এমন ভাবে Block বানানো যাতে ওই Block এর সব Query এর জন্য Time Pointer মোট $O(n)$ Move করে।

MO Without Update এ আমাদের Block ছিল $O(\frac{n}{k})$ টা। আমরা যেই Query গুলার Left Block same তাদের right side based sort করছিলাম।

এখন Update এর জন্য আমরা k size এর block নেব। তাহলে $O(\frac{n}{k})$ টা Block থাকবে।

এখন আমরা Query গুলাকে **Group** করব এইভাবে - "যেই সব Query এর Left side $Block - x$ এ, এবং Right side $Block - y$ তে, তারা একই গ্রুপ এ থাকবে"।

মানে এইরকম -
$Group - 01$: যেই সব Query এর Left side $Block - 0$ তে এবং Right side ও $Block - 0$ তে।
$Group - 02$: যেই সব Query এর Left side $Block - 0$ তে Right side আছে $Block - 1$ তে, ইত্যাদি।

সব গুলা Group কে আবার তাদের Start Block আর End Block এর ভিত্তিতে Sorted Order এ Process করতে হবে।

এখন আমরা এই সব গ্রুপ এর মধ্যে query গুলাকে time based sort করব। তাহলে Worst Case এ আমাদের Time Pointer মাত্র একবার 0 থেকে n এ যাবে, $O(n)$

তাহলে আমাদের Compare Function এইরকম হতে পারে -

```
bool cmp(query &a, query &b) {
  int l1 = a.l / k, l2 = b.l / k,
    r1 = a.r / k, r2 = b.r / k;
  // Left blocks differ, they aren't in same group,
  // first comes who have smaller left block.
  if(l1 != l2) return l1 < l2;
  // So here we have same Left Block, but if Right Blocks aren't equal,
  // then who have right block smaller comes first.
  if(r1 != r2) return r1 < r2;
  // Now we have l1 == l2 and r1 == r2,
  // So both a and b query is in same group,
  // sort them based on time.
  return a.t < b.t;
}
```

Real Part - Complexity Analysis

ধরি, আমাদের k length এর ব্লক ছিল, তাহলে মোট ব্লক $\frac{n}{k}$ টা।

তাহলে Group আছে কয়টা? – $O([\frac{n}{k}]^2)$ টা।

প্রত্যেক গ্রুপ এর জন্য এইবার Complexity Analysis করা যাক।

Group টা $Block-x$ আর $Block-y$ এর জন্য হলে -

Left Pointer খালি $Block-x$ এর মধ্যে সামনে পিছে করে, একটা Query এর জন্য Worst Case এ $O(k)$।
Right Pointer খালি $Block-y$ এর মধ্যে সামনে পিছে করে, একটা Query এর জন্য Worst Case এ $O(k)$
আর একটা Group এর মধ্যে Query গুলা time অনুযায়ী sorted ছিল। তাই ওই group এর সব Query এর জন্য $O(n)$ সরবে।

তাহলে Q টা Query হলে -
Left Pointer মোট সরে - $O(Qk)$
Right Pointer মোট সরে - $O(Qk)$
Time Pointer প্রত্যেক গ্রুপ এর জন্য মোট $O(n)$ সরে। Group আছে $O(\frac{n^2}{k^2})$ টা। তাহলে মোট $O(\frac{n^3}{k^2})$

Total = $O(Qk + \frac{n^3}{k^2})$

$Q = O(n)$ হলে এইটা Minimum হবে যদি $nk = \frac{n^3}{k^2}$ হয়। তাহলে -

$$
\begin{aligned}
nk &= \frac{n^3}{k^2} \\
\implies n^3 &= nk^3 \\
\implies n^2 &= k^3 \\
\implies k &= \sqrt[3]{n^2}
\end{aligned}
$$

তার মানে আমাদের Block Size $k = \sqrt[3]{n^2}$ নিতে হবে।

এক কোথায় বললে বলা যায়, এই Approach Optimal যদি Block Size $k = n^{\frac{2}{3}}$ হয়। তাহলে আমাদের Block থাকবে $\frac{n}{n^{\frac{2}{3}}} = n^{\frac{1}{3}}$ টা।

তাহলে প্রত্যেকটা Query এর জন্য Overall Time লাগবে $O(n^{\frac{2}{3}})$
$Q = O(n)$ হলে মোট $O(n \times n^{\frac{2}{3}}) = O(n^{\frac{5}{3}})$ [Nyc Complexity]

Implementation Practice

Implementation Practice এর জন্য এই প্রবলেম টা Submit করে দেখতে পারেন - HRSIAM (http://www.spoj.com/problems/HRSIAM/).

MO's Algo with Updates Problem গুলা খুবি Rare. আর সাধারণত এরকম সব Problem ই Boss Problem হয়। Ongoing Contest এ লাগলে Div1D/E তে লাগতে পারে। 😛

ধন্যবাদ সবাইকে। 🙂

Category: Data Structure (/category#Data%20Structure)

(htt... (htt... (htt...

← PREVIOUS POST (/POSTS/PERSISTENT-SEGMENT-TREE-02/)

NEXT POST → (/POSTS/FAST-FOURIER-TRANSFORM/)