Adatszerkezetek és algoritmusok 3. kis házi feladat

A feladatok során a bemenetek nem léphetnek túl a limiteken, ezt nem kell külön ellenőrizni.

1 HeavyHitters

„Mese”

A házifeladatok beadásának határidejéhez közeledve az automata háziellenőrző túlterhelte a szervert ezért az nem volt képes több kérést fogadni. A commitokat visszanézve kiderült, hogy az egyik felhasználó random generált stringeket töltött fel megoldás forrásfájlként, mondván, hogy több esélye van a randomgenerátornak a feladat megoldására, mint neki. Ennek kivitelezésére rövid idő alatt rengeteg commit érkezett a felhasználótól és ezek a commitok voltak a felelősek a forgalom túlnyomó többségéért. Annak érdekében, hogy ilyen kiesés ne fordulhasson elő, arra jutottunk, hogy nagy forgalom esetén nem fut le a háziellenőrző azon felhasználók esetén, akik az elmúlt időszakban a commitok túlnyomó többségéért felelősek. Mivel ennek a szűrésnek lényegében real-time kell működnie és a szerveren limitált erőforrások állnak rendelkezésre ezért a commitok számolása nem fenntartható megoldás.

A feladat egy olyan algoritmus implementálása, ami egy N hosszú adatsorból kiszűri azokat az adatokat, amik az adatsor legalább $\frac{1}{k}$-ad részét teszik ki. Fontos kikötés, hogy az algoritmus egy gyors streaming algoritmus legyen $\mathcal{O}(k)$ memóriaigénnyel.

Egy ilyen algoritmus a HeavyHitters (lehetséges más megoldás is, amennyiben teljesíti a követelményeket, de a továbbiakban ezt tárgyaljuk).

Szemléltetés

A HeavyHitters algoritmus megértéséhez vegyük a $k = 2$ esetet. Tegyük fel, hogy van egy szobánk, ahova egyesével lépnek be emberek. Minden emberről ránézésre el tudjuk dönteni, hogy mit használ IDE-nek programozáshoz. (Minden IDE képviselője, meg van győződve róla, hogy igazi programozók csak az általa preferált IDE-ben programoznak és ezért képes vérre menő csatákat folytatni.) Tehát belép az első programozó, aki CLiont használ, ameddig csak CLiont preferáló emberek vannak a teremben teljesen békésen elbeszélgetnek egymással. Amint belép egy ember, aki például nanot használ, ekkor az egyik CLiont preferáló végkimerülésig tartó vitába bonyolódik a vele és elhagyják a termet. Az utolsó érkező után tudjuk, hogy a teremben csak 1 IDE képviselői vannak a teremben (ha lenne 1 másik IDE-t használó bent akkor az már vitát folytatna valaki mással és elhagynák a termet). Összesen maximum $\frac{N}{2}$ vitatkozó pár alakulhat ki, ezért beláthatjuk, hogy ha egy IDE abszolút többségben van az emberek között, akkor nincs olyan sorrend, aminek következtében nem ennek az IDEnek a támogatói maradnak a teremben beszélgetve. Persze az is lehet, hogy az utolsó érkező árva egy (megtévedt) Eclipse hívő és a végén csak őt látjuk a teremben. Magányosan. Tehát anak ellenőrzésére, hogy az algoritmus végén maradt IDE ténylegesen többségi elem, még egyszer végig kell futnunk az eddigi embereken és megszámolnunk, hányan vannak a végső IDE-t támogatók.

Ebből a példából természetesen következik az algoritmus tetszőleges k-ra, az eltérés annyi, hogy nem 2 ember üti ki egymást, hanem k különböző IDEt használó ember.

Algoritmus

Az algoritmus lényegében 2 fázisból áll. Az első fázis során $k-1$ darab érték - számláló párt tartunk számon, és minden egyes érték érkezésekor ellenőrizzük, hogy az érték benne van-e az eddig számon tartott értékek között, ha igen akkor növeljük a hozzá tartozó számlálót, ha nincs és még kevesebb mint $k-1$ értéket tartunk

számon, akkor felvesszük ezt az értéket is 1-es számlálóval, ha $k-1$ másik érték van számon tartva, akkor mindegyiknek csökkentjük a számlálóját (és a 0 számlálóval rendelkezőket eltávolítjuk).
A második fázis a végén megmaradt maximum $k-1$ értékhez tartozó elemek leszámolása. A végső eredmény azon értékek halmaza lesz, amik több mint $\frac{N}{k}$-szor szerepeltek az adatok között.

Részletesebb magyarázat itt.

Limitek

- Adatsor mérete: $1 \leq A \leq 10^8$
- Időlimit: az utolsó tesztesetre 1 másodperc az összes többire tesztesetenként 0.1 másodperc
- Memórialimit: 100 MiB

.

API

A feladat megoldásához implementáld a következő függvényt:

```
template <typename ForwardIterator,
          typename ValueType =
              typename std::iterator_traits<ForwardIterator>::value_type>
std::vector<ValueType> heavy_hitters(ForwardIterator begin, ForwardIterator end,
                                     size_t k)
```

A `ForwardIterator` típusparaméter valamilyen iterátor típus lesz (van neki $++$, $!=$, $==$, $*$ operátora), míg a `ValueType` foglalja magában az iterátorok dereferálásakor kapható objektumok típusát (pl.: `*begin` típusát). Ennek a típusnak a megtudására 2 út létezik, a fenti `iterator_traits` típus által meghatározott `value_type` típusparaméter használata, vagy a `decltype` függvény használata (`decltype(*begin)`), aminek a visszatérési értéke a paraméter típusa. Mi most az előbbit választottuk. A feladat szempontjából elég annyi, hogy az adatsor `ValueType` típusú elemekből fog állni.

2 Range Minimum/Maximum Query

A második feladat egy range minimum query (RMQ) adatszerkezet implementálása (a mi esetünkben maximummal fogunk foglalkozni). Ezt az adatszerkezetet arra tudjuk használni, hogy egy adatsorozaton belül egy egybefüggő résznek a maximumát tudjuk megadni konstans időben. Azaz: legyen $D = \{d_i \in \mathbb{R} | i \in \{1, \ldots, N\}\}$ az adatsor, ekkor $0 \leq L \leq R \leq N, L, R \in \mathbb{N}$ esetén $RMQ\langle D \rangle(L, R) = max_{i \in \{L, \ldots, R\}}\{d_i\}$.

Az adatszerkezet felépítéséhez egy `std::vector<double>`-t kap paraméterül. Ebből építi fel az adatszerkezetet. Az RMQ-n egy művelet van értelmezve és ez a `max`, ami két egész számot kap paraméterül (két indexet) és az ezek által meghatározott tartományban a maximum érték indexével tér vissza.

Az RMQ naív implementációja egy olyan táblázat, amiben minden index párhoz hozzárendeljük a maximum értéket. Ennek a tárhely komplexitása $\mathcal{O}(N^2)$. A feladat általunk elfogadott megoldásához azonban $\mathcal{O}(Nlog_2N)$-re kell leszorítani a felhasznált tárhelyet. Ezt úgy érhetjük el, hogy minden indexhez eltároljuk az onnan induló 2 hatvány hosszú tartományok maximumát. Ekkor azonban a nem kettő hatvány hosszú kérdésekre csak úgy tudjuk megadni a pontos választ, ha az L-től induló és az R-nél végetérő két kettőhatvány hosszú tartományra kapott válaszokat együttesen vizsgáljuk. Példa az 1. táblázatban a felépítednő tömbről.

Limitek

- Adatsor mérete: $1 \leq A \leq 10^7$
- Időlimit: az összes tesztesetre tesztesetenként 0.2 másodperc
- Memórialimit: 100 MiB

.

l	0:1	1:2	2:4	3:8
0	0	1	1	7
1	1	1	1	7
2	2	3	4	7
3	3	4	6	7
4	4	4	7	7
5	5	6	7	7
6	6	7	7	7
7	7	7	7	7
8	8	8	8	8

1. táblázat. RMQ tábla az A = [0, 5, 2, 3, 5, 5, 6, 8, 7] tömbre

API

A feladat megoldásához implementáld a következő osztályt:

```
class rmq {
public:
  rmq(const std::vector<int>&);
  rmq(const rmq&);
  rmq &operator=(const rmq&);
  int max_pos(int left, int right) const;
};
```

Ahol a `max_pos` függvény a paraméterül kapott két index által meghatározott tartományon belül a maximum elem indexét adja vissza.

A copy konstruktor és assignment operátorokat, ha úgy gondolod, hogy helyes implementációt ad `default` kulcsszóval lásd, ha nem akkor megfelelően implementáld. A move konstruktor és assignment operátor-t nyugodtan lehet `delete`-tel megjelölni (de implementálni, vagy defaultolni is ér).