Adatszerkezetek és algoritmusok 4. kis házi feladat

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

1 HyperLogLog

Adott egy A elemszámú adatsor, a feladat az, hogy hatékony becslést tudjunk adni arra, hogy hány különböző elem van benne. Tegyük fel, hogy az adatsor A mérete lehet akkora, hogy nem fér be a memóriába, sőt csupán a különböző elemek sem férnek el a memóriába. Ennek következtében pusztán a különböző adatok leszámolása, vagy fában tárolása már nem elegendő nekünk.

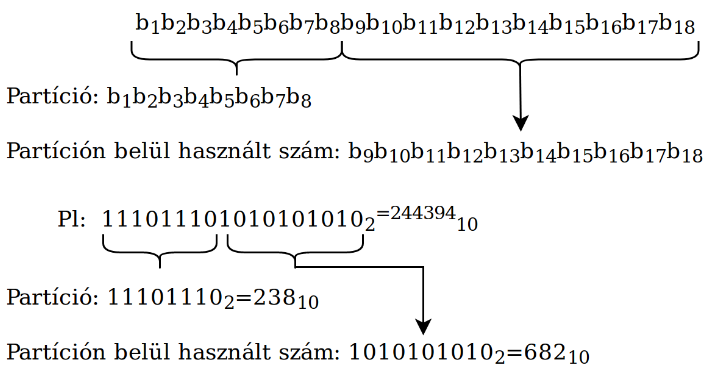
Szerencsére a feladat megoldásához – hosszú keresgélés után – lehet találni egy streaming algoritmust, amit HyperLogLog-nak hívnak.

A streaming algoritmusok jellemzője, hogy úgy kell feldolgozniuk adathalmazokat, hogy kevésszer (általában csak egyszer) futhatnak végig az adatokon és általában kevés memóriát használnak.

A HyperLogLog algoritmus 3 fő megfigyelésre épít:

- Az adatok hash-e tekinthető véletlen számnak
- A véletlen számok esetén K darab kezdő 0 bit valószínűsége $\frac{1}{2^K}$
- A random számok bitmintája felbontható kisebb random számokra:
  - Adott random 4 bites szám: $b_1b_2b_3b_4$, ekkor a b_1b_2 és a b_3b_4 2 bites számok is random számok

A hyperloglog működésének megértéséhez először is vegyünk egy N hosszú random bitsort, amiben $\forall i \in \{1,\ldots,N\}$, $P(b_i = 1) = 0.5$. Ebben az esetben annak a valószínűsége, hogy az első K bit 0: $P(b_0 == 0 \wedge b_1 == 0 \wedge \ldots \wedge b_K == 0) == \frac{1}{2^K}$ Tehát, ha veszünk például 128 különböző ilyen random számot, akkor ezek közül várhatóan egynek lesz legalább 7 vagy annál több 0 bit az elején. Ha ezt megfordítjuk kapunk egy nagyon rossz becslést arra, hogy hány különböző elem volt az adatsorban. Tehát azt becsüljük, hogy 2^k db elem van egy halmazban, ha a kezdő nullák száma maximum k volt. Annak érdekében, hogy a véletlenül érkező sok kezdő nullával bíró számok által hozott hibát csökkentsük a random számok első m bitjét és a maradékot két random számként fogjuk kezelni. Az első m bit meghatároz egy partíciót, a maradék biteken pedig a kezdő nullák számát nézzük. A felbontásra példát az 1. ábrán láthatunk.



1. ábra. Egy 18 bites szám szétválasztása egy 8 bites partíció indexre és egy 10 bites egész számra

Így lényegében $M = 2^m$ részre bontottuk a bemeneti teret (a hash ugyanazat a számot mindig ugyanabba a partícióba küldi). M_i jelölje, ami az i-edik ilyen partícióba érkező számoknál mért kezdő nullák számának maximumát, ekkor a becsélsünk az i-edik partícióba érkezett egyedi számok számára 2^{M_i}. Így 2^m független becslésünk van ugyanennyi kisebb problémára. Ezek után az lehetne a tippünk, hogy ezeket a becsléseket összegezve megkapjuk a végső becslésünket. Okos emberek kitalálták, hogy ez nem elég jó becslés, ezért a mérések az összege helyett a végső becslésünket úgy kapjuk, hogy a harmonikus közepüket szorozzuk meg a partíciók számával:

$$n \approx \alpha_M M^2 \left(\sum 2^{-(M_i+1)} \right)^{-1}$$

Az α értékeire, pedig [1]:

$$\begin{aligned} \alpha_{16} &= 0.673 \\ \alpha_M &= 0.72134/(1 + 1.079/M) \text{ for } M \geq 128 \\ \alpha_{\inf} = \lim_{M \to \inf} \alpha_M &= 0.72134 \end{aligned}$$

A saját implementáció megírásakor használjatok 14 bitet partíciók megjelölésére és az $\alpha_{\inf}$ határértéket, vagy a megfelelő értéket a képlet kiértékeléséből. Részletesebb magyarázat itt.

Limitek

- Adatsor mérete: $1 \leq A \leq 10^9$
- Időlimit: az összes tesztesetre tesztesetenként 0.6 másodperc, kivéve a nagy tesztet ahol 3 márodperc
- Memórialimit: 100 MiB

Megjegyzés: Az adathalmaz 10^9 mérete adatonként 1 db *double* értékkel 7 GiB mérethez vezet.

API

A feladat megoldásához implementáld a következő függvényt:

```
template <typename ForwardIterator>
size_t hyperloglog(ForwardIterator begin, ForwardIterator end);
```

Egyéb információk

A fenti API-ban használt template paraméter mindig, egy szabályosan megírt forward iterátor típus lesz. Ebből következően tudjuk, hogy a dereferálásakor kapott típusra hivatkozhatunk a következő képpen:

```
std::iterator_traits<ForwardIterator>::value_type
```

Illetve hash-elésre használjátok a c++ beépített hash függvényét (általában nem jó ötlet, de mivel most amúgy is minden teszthalmaz random lesz ezért nem fog befolyásolni). A függvény paraméteréül kapott $begin$ iterátor által mutatott értékre például a következő képpen lehet meghívni:

```
std::hash<typename std::iterator_traits<ForwardIterator>::value_type> hash_func{};
size_t hash_value = hash_func(*begin);
```

Ez a függvény egy 64 bites `size_t` típussal tér vissza. Ez lesz a véletlen szám, aminek a felső 14 bitjét a partíció kijelölésére, az alsó 50 bitjét pedig a kezdő nullák számolására kell felhasználni. A kezdő nullák számolására pedig érdemes a `__builtin_clzl` függvényt használni. (Ez a függvény hatékony "CPU kódra" fordul.)

References

[1] S. Heule, M. Nunkesser, A. Hall, Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm, in: Proceedings of the 16th International Conference on Extending Database Technology, ACM, 2013, pp. 683–692 (2013).