Tema 2 - Count-distinct problem

Responsabili:

- Luca Istrate [mailto:lucaistrate@gmail.com]
- Dănuţ Matei [mailto:matei.danut.dm@gmail.com]

Data publicării : 26 martie, ora: 21:00

Deadline: 17 aprilie, ora 23:55

Modificări şi actualizări

- **27 martie, ora 01:30** - adăugat clarificari in sectiunea **Introducere**
- **27 martie, ora 21:05** - adăugat **checker** in **enunt** si pe **vmchecker**
- **27 martie, ora 23:40** - modificat fisier **Makefile** din **checker**
- **29 martie, ora 16:00** - adăugat precizari in **cerinta II** legate de **dimensiunea Hashtable-ului**
- **29 martie, ora 18:40** - adăugat corectare in **cerinta III** legata de **definitia lui m**
- **29 martie, ora 19:30** - modificat fisier **check.sh** din **checker**; acum partea de **valgrind** are comportamentul corect pentru toate cerintele
- **3 aprilie, ora 19:10** - reformulat **cerinta III** pentru a o face mai usor de inteles
- **11 aprilie, ora 12:05** - actualizat **deadline**

Obiective

În urma realizării acestei teme:

- veţi învăţa să lucraţi cu Dictionare
- vă veţi familiariza cu rezolvarea unei probleme reale prin structuri de date din ce in ce mai eficiente

Introducere

Problema estimarii cardinalitatii (a numararii elementelor distincte) este, in esenta, gasirea numarului de elemente unice dintr-o colectie de elemente care se pot repeta.

Pentru cerintele **I** si **II**, vom rezolva o problema si mai restrictiva: gasirea **numarului de aparitii** pentru fiecare element. Pentru cerinta **III**, vom vedea ca acest lucru e mai greu realizabil cand vine vorba de volume mari de date si, de aceea, ne vom rezuma la **gasirea numarului de elemente distincte**.

Conceptual, ne referim la:

INPUT:

```
1, 34, 2, 2, 2, 3
```

OUTPUT:

Pentru cerintele I si II:

```
1 - 1
2 - 3
3 - 1
34 - 1
```

Pentru cerinta III:

```
Exista 4 elemente distincte
```

Fiecare dintre cele **3** cerinte se va implementa intr-un **fisier separat**.

I. Vector de frecventa - 25p

La intrare se dau numere intre **0** si **2000000**. Gasiti numarul de aparitii ale fiecarui element, utilizand un **vector de frecventa**.

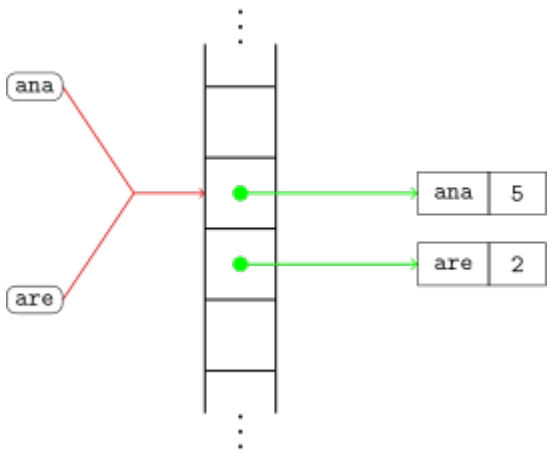
Un vector de frecventa este un vector care are pe pozitia **i** *numarul de aparitii* ale elementului **i**.

*Se garanteaza ca numarul de aparitii ale oricarui element este mai mic decat **256**.*

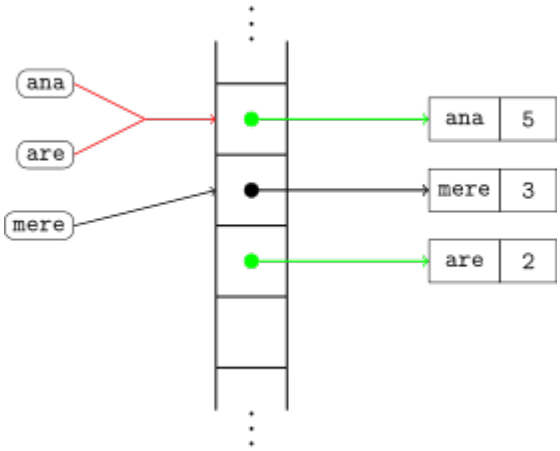
II. Hashtable cu open addressing - 25p

La intrare se dau siruri de caractere. Gasiti numarul de aparitii ale fiecarui sir folosind un Hashtable cu politica de rezolvare a conflictelor de tip **open addressing** prin **linear probing**.

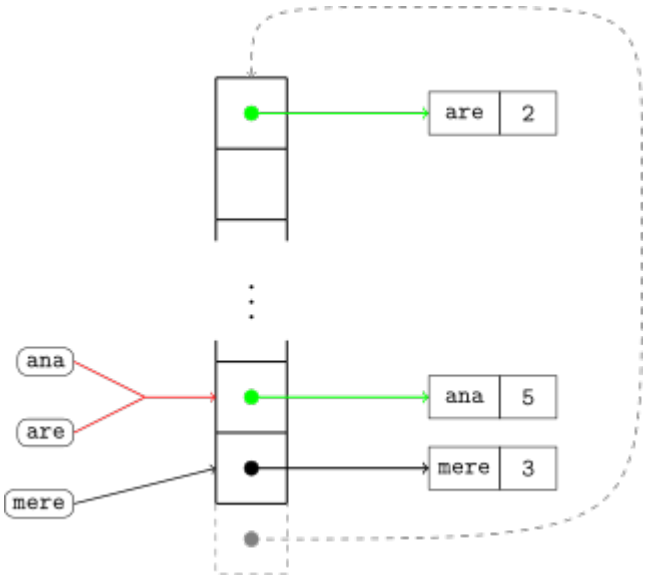
Aceasta politica presupune ca, in momentul in care bucketul unde trebuie realizata insertia este deja ocupat, *se va cauta secvential o pozitie libera incepand cu bucketul urmator*.



Evident, daca si aceasta pozitie este ocupata, vom cauta prima pozitie libera in continuare.



Daca se va ajunge la finalul listei de bucketuri, se va continua de la inceput.



Se garanteaza existenta a cel putin unui bucket liber in momentul fiecarei operatii de insertie. **Pentru a satisface aceasta conditie, o idee ar fi ca dimensiunea Hashtable-ului sa fie egala cu numarul de siruri existente in fisierul de intrare.**

Evident, daca in momentul unei operatii de selectie nu gasim cheia in bucketul in care ne-am astepta, vom continua cautarea secvential, aplicand un procedeu similar cu cel din momentul insertiei.

*Se garanteaza ca lungimea maxima a oricarui sir este maxim **100** de caractere.*

*Se garanteaza ca numarul de aparitii ale oricarui element este mai mic decat **256**.*

III. Estimatori probabilistici - 30p

In cerintele anterioare am observat ca putem calcula cu exactitate numarul de elemente distincte (si numarul lor de aparitii) retinand, intr-un fel sau altul, fiecare element unic (ca pozitie intr-un vector, respectiv ca cheie intr-un hashtable). Din pacate, in aplicatiile din lumea reala, *aceasta strategie nu este sustenabila*.

Sa ne imaginam urmatoarea situatie: **Youtube** afiseaza pentru fiecare videoclip **numarul de vizualizari**. Pentru ca acest sistem sa nu fie abuzat (spre exemplu de un bot care vizioneaza acelasi videoclip incontinuu pentru a-i spori view count-ul), trebuie sa tina cont si de **numarul de utilizatori unici** care au accesat clipul.

Daca ar retine un id pentru fiecare utilizator, ar avea nevoie de o structura de date cu milioane de intrari, si asta doar pentru un singur clip. Tinand cont ca exista *peste **31 de milioane** de canale* (iar multe dintre ele au **peste 100 de clipuri**), acest lucru iese din discutie.

Cum putem totusi sa ne indeplinim obiectivul?

Problema la abordarea anterioara este faptul ca foloseste o cantitate de memorie proportionala cu numarul de utilizatori distincti **O(n)**. In cautarea unei solutii mai bune, va trebui sa obtinem o complexitate a spatiului mai mica (**O(sqrt(n))**, **O(logn)**, **O(1)** etc.)

Solutia gasita (ce urmeaza a fi implementata in tema) aduce cu sine un sacrificiu din punct de vedere al preciziei: din moment ce nu stim exact ce utilizatori am avut, numarul total de utilizatori unici **nu va mai fi unul precis, ci doar aproximativ**.

In practica, acest lucru nu este un dezavantaj prea mare (intrucat rareori e relevanta diferenta dintre 1m vizualizari si 1.1m vizualizari).

In ilustrarea functionarii algoritmului **HyperLogLog**, vom incepe de la o serie de principii simple pe care le vom pune cap la cap, ajungand la descrierea algoritmului final.

Sectiunile 1 si 2 sunt prezentate pentru a intelege de ce functioneaza HyperLogLog. Pentru a rezolva tema, trebuie sa implementati **doar algoritmul final (descris in sectiunea 3)**.

1. Probabilistic counting

Sa presupunem ca generam un numar la intamplare.

Probabilitatea ca numarul sa inceapa cu **un** bit **0** este **1/2** (deoarece poate incepe fie cu 0, fie cu 1).

Probabilitatea ca numarul sa inceapa cu **2** biti **0** este **1/4** (deoarece poate incepe cu 00, 01, 10 sau 11).

Similar, probabilitatea ca numarul sa inceapa cu **3** biti **0** este **1/8**. Astfel, pentru a intalni un numar care sa inceapa cu **3** biti **0**, *va trebui sa generam, in medie, **8** numere*.

$$\frac{1}{2}\frac{1}{2}\frac{1}{2} = \frac{1}{8}$$

```
0b00011010
0b10101100
    ⋮
0b11110101
```

Privind aceasta observatie in sens invers, daca am generat numere aleatoare si secventa cea mai lunga de **0** de la inceputul oricarui numar a fost de lungime **3**, atunci avem doua posibilitati:

- am generat *cel putin **8** numere*

- am avut noroc si a trebuit sa generam *mai putin de 8 numere*

Evident, pentru valori mici precum **2** sau **3** biti consecutivi, exista o sansa semnificativa sa generam numarul mai rapid (chiar din prima incercare), dar cu cat valorile devin mai mari, cu atat scade aceasta sansa.

In principiu, daca am primit valori aleatoare la intrare, si numarul maxim de biti **0** consecutivi initiali ai oricarui numar este **x**, putem spune ca am primit *intre **2^x** si **2^(x+1)** valori*.

In viata reala, valorile primite la intrare *nu vor fi neaparat valori aleatoare*. Mai mult, nu toate valorile de intrare vor fi numere intregi. Din acest motiv, vom trece aceste valori printr-o **functie de hash**. O functie de hash buna ar trebui sa ofere la iesire *valori uniform distribuite*.

2. LogLog

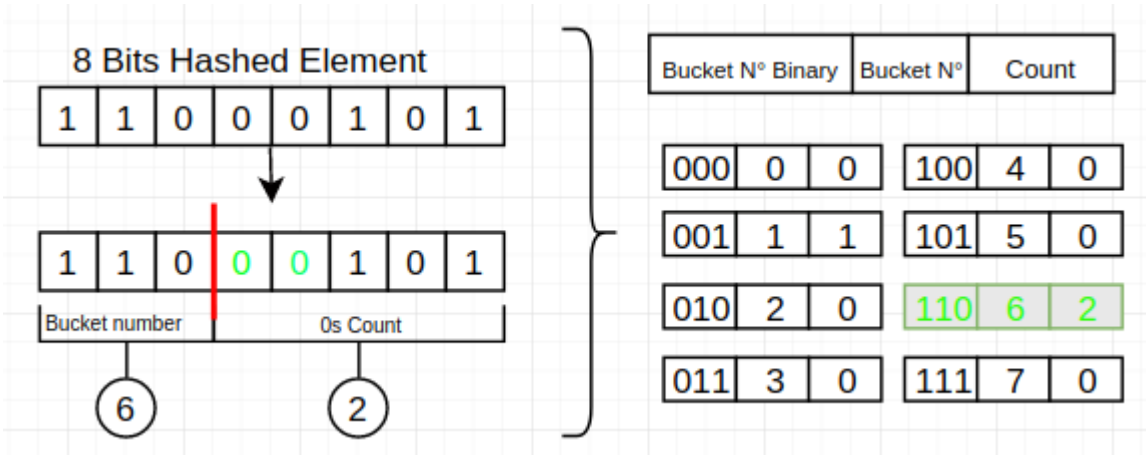
Daca vrem sa imbunatatim performanta algoritmului nostru va trebui sa:

- Atenuam efectul negativ al generarii rapide unui numar cu multi biti de **0** initiali

- Oferim estimari *mai granulare* decat **puterile lui 2**

2^0	**1**
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024
2^{11}	2048
2^{12}	4096
2^{13}	8192
2^{14}	16384
2^{15}	32768
2^{16}	65536

O idee de rezolvare a primei probleme este sa impartim numerele in mai multe **bucketuri**. Cea mai usoara modalitate de a face acest lucru este sa impartim fiecare numar in 2 parti: *prima parte* va fi folosita pentru a determina bucketul, iar *a doua* va fi folosita ca pana acum.



Singura diferenta fata de metoda precedenta este ca acum vom face maximul de zerouri consecutive *pentru fiecare bucket* si nu pentru toate numerele.

Pentru a agrega aceste maxime vom folosi **media geometrica**.

3. HyperLogLog

Recapituland ce am prezentat in sectiunile precedente, in cadrul algoritmului HyperLogLog avem 3 etape:

1) stabilim numarul total de bucketuri **m**, apoi initializam cu 0 un vector **M** de dimensiune **m**.

Alegerea lui m este diferita de la caz la caz. In contextul problemei curente, puteti sa folositi valoarea m = 2^{11}, insa exista si alte variante posibile.

2) pentru fiecare numar citit de la intrare:

- ii calculam hash-ul cu o functie de hash pentru numere intregi

- pe baza primilor $\log_2(m)$ biti din hash determinam bucketul in care se afla (din cele m bucketuri posibile); notam numarul bucketului cu **j**

- calculam numarul de biti 0 initiali din restul hash-ului; notam acest numar cu **x**

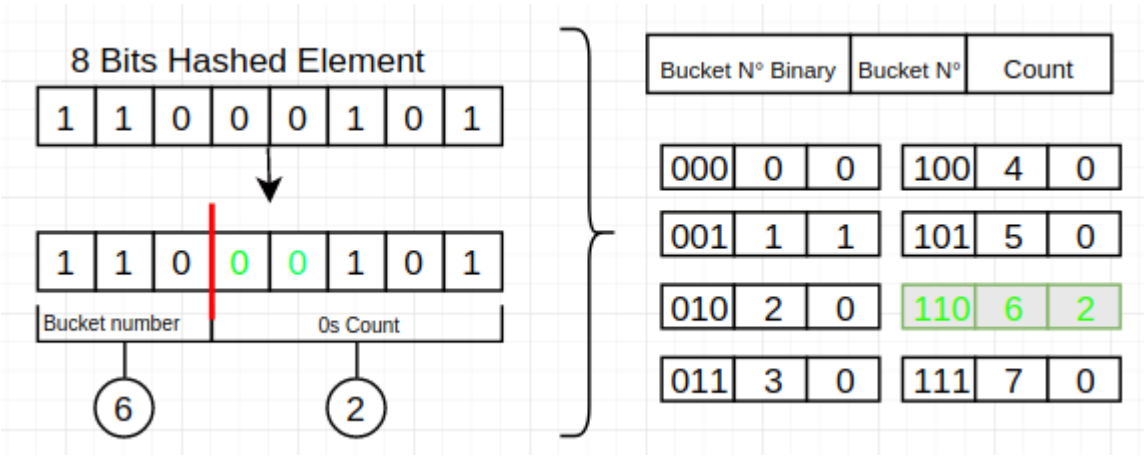
- M[j] = max(M[j], x)

3) agregam valorile din toate bucketurile

In sectiunea precedenta, am mentionat ca pentru a agrega valorile din fiecare bucket, folosim media geometrica. Pentru a implementa HyperLogLog, vom folosi in locul ei urmatorea medie:

$$Z = \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

Ca exemplu, pentru bucketul evidentiat cu verde, **j = 6**, iar **M[j] = 2**



Avand aceasta medie **Z**, raspunsul final **E** (numarul de elemente distincte intalnite) va fi dat de urmatorea formula:

$$E = \alpha_m m^2 Z$$

Explicatie:

m, ca si pana acum, este numarul total de bucketuri folosite

α_m reprezinta *factorul de atenuare*, calculat in functie de **m** dupa urmatoarea formula:

$$\alpha_m \approx \begin{cases} m = 16 & 0.673 \\ m = 32 & 0.697 \\ m = 64 & 0.709 \\ m \geq 128 & \frac{0.7213}{1+\frac{1.079}{m}} \end{cases}$$

Pentru ultima cerinta, citirea se va face dintr-un fisier al carui nume este primit ca parametru.

Precizări

Rezolvati cerintele I si II utilizand structura de date ceruta. Nerespectarea acestui lucru va aduce la anularea punctajului pentru cerinta respectiva.

Având în vedere ca a 3-a parte a temei presupune implementarea unei structuri de date probabilistice, checkerul ofera punctajul daca raspunsul vostru se incadreaza intr-o marja de eroare de 10% fata de raspunsul corect.

Checker

CHECKER

Temele vor fi trimise pe vmchecker [https://elf.cs.pub.ro/vmchecker/ui/#SD]. **Atenție!** Temele trebuie trimise în secțiunea **Structuri de Date (CA)**.

Arhiva trebuie să conțină:

- surse
- fișierul Makefile **din arhiva cu checkerul**
- fișier **README** care să conțină detalii despre implementarea temei

Punctaj

Atentie! O temă care nu compilează va primi 0 puncte.

1. 80p teste
2. **Fiecare** test este verificat cu valgrind. Dacă un test are memory leaks, nu va fi punctat.
3. 20p README + comentarii/claritate cod (ATENȚIE! Fișierul README trebuie făcut explicit, cât să se înțeleagă ce ați făcut în sursă, dar fără comentarii inutile și detalii inutile).
4. Se acordă 20% din punctajul obținut pe teste, ca bonus pentru coding style. De exemplu, pentru o temă care obține maxim pe teste, se pot obține 20p bonus dacă nu aveți erori de coding style. Pentru o temă ce trece 18 teste din 20, se pot obține 18p dacă nu aveți erori de coding style.
5. O temă care obține 0p pe vmchecker este punctată cu 0.
6. Temele au deadline hard. Prin urmare, o temă trimisă dupa deadline este punctată cu 0.

Nu copiați! Toate soluțiile vor fi verificate folosind o unealtă de detectare a plagiatului. În cazul detectării unui astfel de caz, atât plagiatorul cât și autorul original (nu contează cine e) vor primi punctaj 0 pe **toate temele**!

De aceea, vă sfătuim să nu vă lăsați rezolvări ale temelor pe calculatoare partajate (la laborator etc), pe mail/liste de discuții/grupuri etc.

FAQ

Q: Ce functii de hashing trebuie sa folosesc in tema, la cerintele II si III?

A: Puteti folosi orice functii doriti. Un exemplu ar fi cele din laborator.

Q: La cerinta II functia mea de hashing nu imi genereaza deloc coliziuni. E ok?

A: E in regula, insa codul care trateaza posibilitatea coliziunilor **trebuie sa existe**.

Q: In enuntul cerintei III sunt mentionate functiile matematice log si pow, insa checkerul nu permite folosirea functiilor matematice. Cum rezolvam problema asta

A: Pentru a-l calcula pe **m** care e de forma 2^k, puteti folosi shiftarea pe biti, adica

```
int m = 1 << k;
```

Din moment ce k se stabileste in prealabil, $\log_2 m = k$.

Link-uri utile

https://en.wikipedia.org/wiki/Count-distinct_problem [https://en.wikipedia.org/wiki/Count-distinct_problem]

https://www.omnicoreagency.com/youtube-statistics [https://www.omnicoreagency.com/youtube-statistics]

https://en.wikipedia.org/wiki/HyperLogLog [https://en.wikipedia.org/wiki/HyperLogLog]

https://en.wikipedia.org/wiki/Linear_probing [https://en.wikipedia.org/wiki/Linear_probing]

https://stackoverflow.com/questions/12327004/how-does-the-hyperloglog-algorithm-work [https://stackoverflow.com/questions/12327004/how-does-the-hyperloglog-algorithm-work]