École Polytechnique — CSE202 Design and Analysis of Algorithms

Midterm Exam

Estimating the cardinality of a list using small memory

Guidelines. Download `loglog.py` and `test_loglog.py`. Write your solution in `loglog.py`. To test your code, run the file `test_loglog.py`. You don't have to justify the complexity of your programs.

The *cardinality* of a list of elements is defined as the number of *distinct* elements that appear in the list. For instance the cardinality of the list `["aa", "ab", "a", "aa", "b", "ab"]` is 4, because there are 4 distinct elements appearing in the list, which are `"a","aa","b","ab"`.

Question 1 (2 points). Complete the function `cardinality(L)` that has to return the cardinality of `L`. Your function should perform a loop on `L` and use an auxiliary structure to store the distinct elements. The complexity should be linear with respect to the length of L.

A disadvantage of the above algorithm is to store the distinct elements, which can be very expensive for the memory [1].

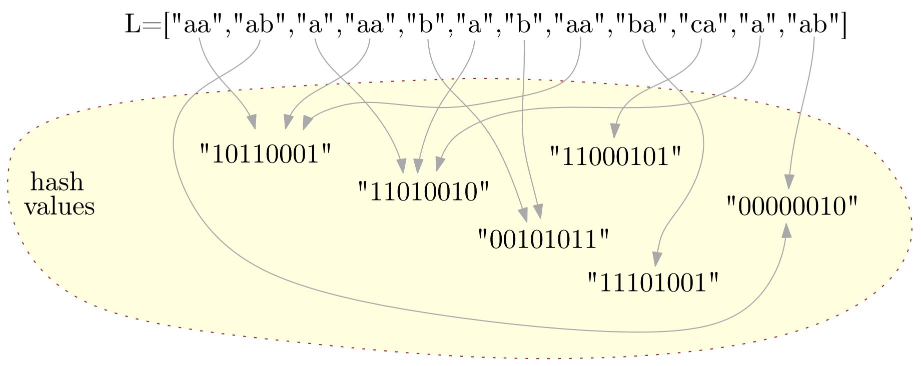
In the next questions we are going to implement a nice probabilistic [2] algorithm that makes it possible to output an estimate (typically correct up to a few percents) of the cardinality, using small auxiliary memory (a table of 1024 small integers will be enough to obtain a precision of 3%, whatever the cardinality of the input list!).

Idea of the algorithm. The crucial ingredient is a hash function that associates a binary string $h(x)$ (here, on 32 bits, given in the function `bin_hash`) to any element x in the input list L. If L has n distinct elements, then there are n different hash values over the whole list (indeed, since h is deterministic, if a distinct element occurs multiple times in L, then each occurence will give the same hash value), and if the hash function has good statistical properties, then about 1/2 of these strings start with 1, about 1/4 of these start with 01, about 1/8 of these start with 001 etc. If we let $f(x)$ be the length of the initial run of zeros in $h(x)$, and define the *size-indicator* s of L as the maximum of $1 + f(x)$ over all elements x in L, we thus expect that 2^s should give a rough indication on the cardinality of L. Unfortunately in this form the estimate is very unprecise. The second idea is to use the b inital bits of the hash values (for some fixed $b \geq 1$) to split the stream of elements of L into $m = 2^b$ *buckets*, compute the size-indicator in each bucket (as illustrated later in Figure 1), and then take some conveniently adjusted average of the m size-indicators as the output cardinality estimate.

Question 2 (2 points). A first step is to complete two functions taking as parameters a bit-string `bina` and an integer $b \in [1..n-1]$, with n the length of `bina` (beware that `bina` is a string whose entries are *char* `'0'` or `'1'`, not numbers; the ith entry can be accessed as `bina[i]`).

The *b-bucket index* of `bina` is defined as the integer corresponding to the leftmost b symbols of `bina`. On the other hand, the *b-zero-length* of `bina` is defined as the largest integer $k \in [0..n-b]$ such that all entries in `bina` from position b to position $b+k-1$ are `'0'` (in particular, the b-zero-length is 0 if `bina[b]` is `'1'`, and is $n-b$ if all the $n-b$ entries of `bina` starting from position b are `'0'`). For instance, if `bina="1100010"` and $b = 3$ then the subtring formed by the first 3 entries is `"110"`, hence the b-bucket index of `bina` is $1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 6$, and the suffix formed by the remaining entries (starting from position $b = 3$) is `"0010"`, hence the b-zero-length of `bina` is 2 (indeed the suffix starts with two `'0'`'s followed by a `'1'`).

1. In practice such algorithms are used to monitor traffic, the list `L` can be a huge stream of arriving data and one would like to answer questions such as "how many distinct connections was there over the last hour". It is thus desirable to avoid storing explicitly all these distinct connections.
2. Actually it is deterministic, but takes advantage of the fact that hash values 'look like' random bit-strings.



buckets for $b = 2$	B_0: "00 000010" "00 101011"	B_1: $\varnothing$	B_2: "10 110001"	B_3: "11 010010" "11 101001" "11 000101"
max value of zeros(·,2)	4	None	0	3

sketch(L,2)=[5,0,1,4]

FIGURE 1 – A list L of elements (top) of cardinality n yields n different hash values, that are redirected into $m = 2^b$ possible buckets $B_0, \dots, B_{m-1}$ according to the leftmost b bits (for a fixed chosen b, here $b = 2$). Each bucket has a size-indicator that is 0 if the bucket is empty, and is 1 plus the max of the zero-lengths (for the same b) over all hash values in the bucket. The function `sketch(L,b)` has to return the array (Python list) whose ith entry is the size-indicator for the ith bucket.

Complete the two functions `bucket(bina,b)` and `zeros(bina,b)` that have respectively to return the b-bucket-index of `bina` and the b-zero-length of `bina` (indication : a bit-string s is converted to an integer using the Python command `(int)(s,2)`, and if `s` is a string, then `s[i:j]` is the substring of characters from position i to position $j-1$).

Question 3 (2 points)**.** For an element x we let $h(x)$ be its hash value (here a bit-string of length 32, you *have to use the function* `bin_hash` given at the beginning of `loglog.py`). For a fixed $b \geq 2$, the *bucket-index* of x is defined as the b-bucket-index of $h(x)$, and the zero-length of x is defined as the b-zero-length of $h(x)$. If we have a list `L` of elements (possibly with repetitions, as in the introductory example of cardinality 4), then for $i \in [0..2^b - 1]$ the *size-indicator* at index i is defined as 0 if `L` has no element of bucket-index i, and otherwise it is one plus the maximum of the zero-lengths over all entries of `L` having bucket-index i.

Complete the function `sketch(L,b)` that takes as arguments a list `L` of elements and an integer b, and has to output (see Figure 1 for an illustration) a list `res` of length $m := 2^b$ such that `res[i]` gives the size-indicator at index i, for each $i \in [0..m-1]$.

Question 4 (2 points)**.** Letting $m = 2^b$ be the number of buckets, n the cardinality of `L`, and $s(i)$ the size-indicator of index i (for $i \in [0..m-1]$), the number $2^{s(i)}$ gives a rough indication of the number of distinct elements with bucket-index i, which should be close to n/m. Thus, we expect that by taking some average of these quantities $2^{s(i)}$, and then multiplying by m, one should obtain a value that approximates n (the larger the m the better the approximation). It turns out that the harmonic mean [3] is the best suited here. Precisely, the cardinality estimator we adopt is

$$E = \mathrm{Const}(b) \cdot \frac{m^2}{\sum_{i=0}^{m-1} 2^{-s(i)}},$$

3. The classical mean of $(x_1, \dots, x_k)$ is $\frac{1}{k}\sum_{i=1}^{k} x_i$, whereas their harmonic mean is $k/\sum_{i=1}^{k}\frac{1}{x_i}$, i.e., is the inverse of the classical mean of $1/x_1, \dots, 1/x_k$.

where $\mathrm{Const}(b)$ is a constant (found by a mathematical analysis not detailed here) that is ≈ 0.673 for $b = 4$, is 0.697 for $b = 5$, etc. (the function `constant(b)` that returns $\mathrm{Const}(b)$ is given in `loglog.py`).

Complete the function `loglog(L,b)` that has to return the estimated cardinality E of `L` as given by the formula above.

Question 5 (2 points)**.** It turns out that the heuristic 'the larger the m the better the approximation' is only valid as long as $n >> m$. When n is not large compared to $m := 2^b$ then the precision of the harmonic-mean estimator becomes quite bad. To remedy this, one can take advantage of the fact that some buckets will be empty, and how many empty buckets there are gives a precious indication on n (the smaller n is compared to m, the more empty buckets).

Precisely the strategy is the following. If the computed estimate E is strictly larger than $5m/2$, then we return E. Otherwise we compute the number V of empty buckets (among the m buckets). If V is non-zero we return $m \log(m/V)$ (neperian logarithm, in Python `math.log`), otherwise we return E.

Complete the function `loglog_small_range_correction(L,b)` that returns the estimated cardinality of `L` taking into account the small-range correction strategy described above.