RANGE UPDATES

Consider an array C of n integers, initially all equal to zero. We want to support the following operations:

- update(i, j, c), where $0 \leq i \leq j \leq n \leq 1$ and c is an integer: it changes C such that $C[k] := C[k] + c$ for every $i \leq k \leq j$.
- query(i), where $0 \leq i \leq n \leq 1$: it returns the values of$C[i]$.
- sum(i, j), where $0 \leq i \leq j \leq n \leq 1$: it returns $\sum_{k=i}^{j} C[k]$

Design a data structure that uses O(n) space and implements each operation above in O(log n) time. Note that query(i) = sum(i, i) but it helps to reason. [Hint to further save space: use an implicit tree such as the Fenwick tree (see Wikipedia).]

SOLUTION WITH RANGE TREE

Given the problem, in which we want to efficiently (log. time, and linear space) work with values stored in an array contained within an arbitrary range $[1...n]$, we need to implement:

- Updating a portion of the range by adding a value c: Update(i, j, c).
- Querrying the value on a given index: Query(i).
- Obtaining the sum of all elements within a range: Sum(i, j).

A possible solution is to model the range as trees, with branches partitioning the overall ranges in smaller ones to efficiently operate on them. Leaves store discrete indices while their parents represent the index ranges contained under them. A tree for $n = 8$, with indices $[0...7]$ is shown below:

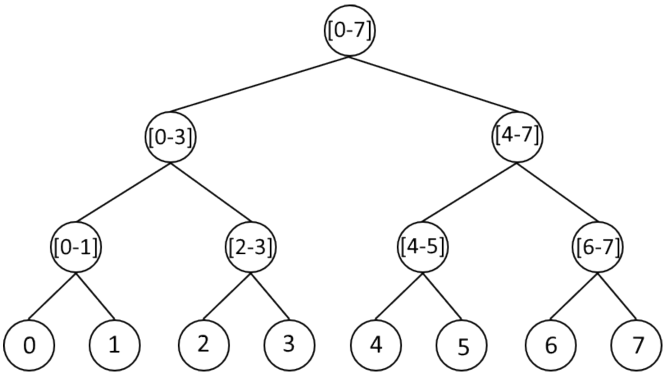


Figure 1: Tree for N=8

Space-wise, this means that for an array of length n, we will at most require $\sum_{i=0}^{\infty} \frac{n}{2^i}$ nodes in the tree, as $\frac{n}{2}$ parents will be needed for the n leaves, $\frac{n}{4}$ for those parents and so on. This geometric series converges to $2n$, leaving us a space complexity bound of $O(n)$.

The implementation of 1-value query is simple: we use the intervals to binary-search our target node. In doing so, we traverse only one tree branch, with time complexity $O(log\ n)$.

Updating and range-summing, on the other hand, are trickier. For them to be efficient, we need to process a whole interval in $O(log\ n)$. To do so, we perform the same binary-search process as before, stopping when the current interval matches the interval of the node. Otherwise we further splitting the interval towards their sub-trees. On a given tree, such splitting will happen at most on $log(n) - 1$ times on both interval extreme, when the range covers all but there is an extreme of the interval that is not fully included. When the intervals match, the marking operation will be applied.

Finally, to efficiently implement both range operations, branch nodes that represent intervals will require two variables: the amount that is added in common to all of their successors and the total sum of their values. That way, operations onto a given node will remain $O(1)$, with our previous analysis being applicable translating into a time complexity bounded by $O(log\ n)$.

SOLUTION WITH BIT TREE
The solution can be implemented using a Segment tree or a Fenwick Tree. Both allowed the aforementioned operation in O(log n). Here we will present a solution that use Fenwick Trees also called Binary Indexed Tree (BIT).
Let's start with some considerations:

- integer can be represented as sum of powers of two. Therefore, I can represent the sum operation as sum of sets of sub-sums.

- each index, if it is a power of 2, will store the sum of all elements before that, and we will apply this repetitively so as to get what each index will store. Let's see an example:
  Suppose, we have an array of 16 elements, $[1 \dots 16]$. Powers of 2 are the index: 1, 2, 4, 8, 16. These index will store sum of all elements before them. Now, we divide this array in two halves: we get $[1 \dots 8]$ and $[9 \dots 16]$, and so on.
  Let's have the following array:

  $$\begin{aligned} A &= [1,2,3,4,5,6,7,8] \\ BIT &= [0,0,0,0,0,0,0,0] \end{aligned} \tag{1}$$

  Let take the power 2 index (i.e. 1,2,4,8), and we store in BIT the value of the sum of the previous element. Then we have:

  $$BIT = [1,3,0,10,0,0,0,36]$$

  Now we divide the array in two: $A_1 = [1,2,3,4]$ and $A_2 = [5,6,7,8]$. Then we take the power 2 index of A_1 and A_2 (i.e. 1,2,4), and we update the value of BIT where there index that are not updated yet. Then we have:

  $$BIT = [1,3,0,10,5,11,0,36]$$

  Following the same procedure, the value doesn't change at any index if it has been already filled. Then we will the following.

  $$BIT = [1,3,3,10,5,11,7,36]$$

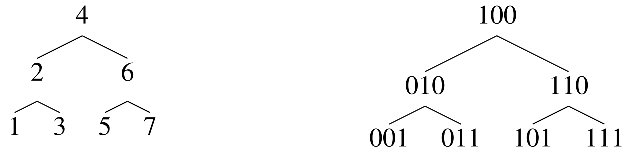
  If we consider our array as a binary tree, and we change the value of each node by adding the sum of nodes in its left sub-tree, we will obtain the same result.

- Now let's see how to implement sum(i, j). The idea is to keep a variable ans initialized to 0. Follow the path from $root$ to the $index$ node. Whenever we need to follow a right link, add the value of current node to ans, and once we reach the node with the searched index we add that value too. To get the sum of elements in range i to j, we get the sum from 0 to j and we subtract the sum from 0 to (i-1).

- Now let's see how to implement update(i, j, c). For now we focus on just the update of a single index. If we want to increment the value at index k by say c. Follow the path from root to the index node k. Whenever we need to follow a left link, add the value of c to current node. Once we reach the node, add c to that node too. This is because we will need to update the set of nodes in the tree that include that node in its left sub-tree, so that it will be consistent with our sum operation.

To implement this kind of idea we need to exploit some properties of the binary numbers. Since for the sum function we need the right-path we exploit the following fact: given an index if we reset the right-most

SET-bit we will go up to the least node that took a RIGHT path. For example:



$$5 \longrightarrow 101 \longrightarrow 100 \longrightarrow 4$$
$$7 \longrightarrow 111 \longrightarrow 110(6) \longrightarrow 100 \longrightarrow 4$$
$$3 \longrightarrow 011 \longrightarrow 010 \longrightarrow 2$$

To implement this concept in to an algorithm we use the following formula $i - (i\ AND(-i))$, where i is the current index and $(-i)$ mean the 2's complement of i.

```
function GETSUM($index$)
    $ans \leftarrow 0$
    while $index\ > 0$ do
        $ans \leftarrow ans + BIT[index]$
        $index \leftarrow index - (index\ AND(-index))$
    end while
    return $ans$
end function
```

Now we should try to do the same for the updating function. Now given an index we should search the least node that took the LEFT path. Here, instead of stripping off the least-significant 1 bit (i.e. subtracting), we now add it on at each stage to get the next entry to adjust. Therefore we simply have:

```
function UPDATE($index$,$val$)
    while $index\ < length\_of\_array$ do
        $BIT[index] \leftarrow BIT[index] + val$
        $index \leftarrow index + (index\ AND(-index))$
    end while
    return $ans$
end function
```

Notice that getSum and update work in O(log n), where n is the length of the input. However, the update function modify one element at the time, then we still need to find a way to do a range update in O(log n). The idea is to keep two BIT (i.e. BIT1 and BIT2) and then modify the functions to sum and update a range.

Let's start with the update function update(i, j, c). When we update BIT1 we will use the previous update as follow:

$$\begin{aligned} update(i, c) &\qquad \textit{\#in BT1} \\ update(j + 1, -c) &\qquad \textit{\#in BT1} \end{aligned}$$

The idea is that the $update(i, c)$ will affect all $i' \leq i$, i.e all the left subtree where we store the partial sum of the sub-interval. To limit the effect to a given range $[i, ..., j]$, we subtract $-c$ from all $i' > j$ by performing the operation $update(j + 1, -c)$. Now, if we use the getSum function as it is, we will have a wrong result. Indeed we should find a way to tell how much the results have to be adjust, here comes the second BIT (i.e BT2). Consider a range $update(i, j, c)$ and let all the elements at the beginning were 0. Now, let consider a getSum(p) for a generic p, then we have:

$$\begin{aligned} &1 \leq p < i \longrightarrow 0 \\ &i \leq p \leq j \longrightarrow c * (p - (i - 1)) \\ &j < p \leq N \longrightarrow c * (j - (i - 1)) \end{aligned}$$

Thus, for a given index p, expanding those formula we obtain the term needed to adjust the prefixed sum. Indeed we find a term X in this way:

$$
\begin{aligned}
&1 \le p < i \longrightarrow 0 \longrightarrow X = 0\\
&i \le p \le j \longrightarrow c * p - c * (i - 1) \longrightarrow X = c * (i - 1)\\
&j < p \le N \longrightarrow c * j - c * (i - 1) \longrightarrow X = -c * j + c * (i - 1)
\end{aligned}
$$

This extra factor keep track of the adjustment needed to obtain the correct sum. To maintain this extra factor X, we use another BIT(i.e BT2) then we have:

$$
\begin{aligned}
update(i, c * (i - 1)) \qquad &\textit{\#in BT2}\\
update(j + 1, -c * j) \qquad &\textit{\#in BT2}
\end{aligned}
$$

Notice $c * (i - 1)$ it's not there because it has been included in the previous update. Indeed to obtain a correct range sum we have to do the following:

$$getSum(p) = getSum(BIT1, p) * p - getSum(BIT2, p)$$

Finally we have the function requested in the exercise:

- $update(i, j, c)$ is equivalent to the following function:
  - $update(BIT1, i, c)$
  - $update(BIT1, j + 1, -c)$
  - $update(BIT2, i, c * (i - 1))$
  - $update(BIT2, j + 1, -c * j)$
- $query(i) = getSum(i) - getSum(i - 1)$
- $sum(i, j) = getSum(j) - getSum(i - 1)$

Both sum, query and update work in O(4 log(n)). This because: the tree is always balanced and the two function they simply take a path in the tree. Furthermore, this algorithm take just O(2 n) space since it's just storing a flat array.[1]

[1] Some references: LINK1, LINK2, LINK3, LINK4 and LINK5

Depth of a node in a random search tree

A random search tree for a set S can be defined as follows: if S is empty, then the null tree is a random search tree; otherwise, choose uniformly at random a key $k \in S$: the random search tree is obtained by picking k as root, and the random search trees on $L = \{x \in S : x < k\}$ and $R = \{x \in S : x > k\}$ become, respectively, the left and right subtree of the root k. Consider the randomized QuickSort discussed in class and analyzed with indicator variables [CLRS 7.3], and observe that the random selection of the pivots follows the above process, thus producing a random search tree of n nodes. Using a variation of the analysis with indicator variables, prove that the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly 2 $ln\ n$.
Prove that the probability that the expected depth of a node exceeds $c\ 2\ ln\ n$ is small for any given constant $c > 1$. [Note: the latter point can be solved after we see Chernoff's bounds.[1]]

SOLUTION
To give an estimation on the depth of a given node, i, we would need to consider how many of the other nodes are its ancestors. For this, an indicator variable can be defined as follows:

$$X_{ij} = \begin{cases} 1 & \text{IF } j \text{ IS AN ANCESTOR OF } i, \\ 0 & otherwise \end{cases}$$

With this indicator variable, analysis can be performed taking the indices as those of the sorted set when in order: $z_1, z_2, ..., z_n$. For two arbitrary indices z_i, z_j only three possible scenarios apply:

1. z_i was selected as a key on the tree before z_j, so z_j is a successor of z_i and thus $X_{ij} = 0$.

2. Neither z_i nor z_j were selected as a key on the tree before, so a key in the range $(i, ..., j)$ or $(j, ..., i)$ effectively splits the range and thus $X_{ij} = 0$.

3. z_j was selected as a key on the tree before z_i, so z_i will eventually be found and selected as a key preceded by z_j and thus $X_{ij} = 1$.

With this analysis in mind, the expectation of the indicator variable can be computed as follows:

$$E[\sum_{\substack{j=1 \\ \text{æ}\neq i}}^{n} X_{ij}] = \sum_{\substack{j=1 \\ \text{æ}\neq i}}^{n} P(X_{ij} = 1) = ...$$

Given the previous analysis, the probability of X_{ij} in the interval containing z_i and z_j on both extremes is that of the only case in which z_j may be an ancestor of z_i over the total amount of cases. In this case that means the number of elements in the range:

$$... = \sum_{\substack{j=1 \\ \text{æ}\neq i}}^{n} \frac{1}{|j - i| + 1} = ...$$

To explicitly take into account the two orderings between z_i and z_j, we split the computation in two terms. The result resembles two instances of the harmonic series, which are then approximated to $ln(n)$:

$$... = \sum_{j=1}^{i-1} \frac{1}{i - j + 1} + \sum_{j=i+1}^{n} \frac{1}{j - i + 1} < ln(n) + ln(n) = 2ln(n)$$

Finally, the depth of a node in a random search tree is expected to be $2ln(n)$. A variation on the same analysis can be performed to estimate the size of the subtree spanning from a given node. In this case, a similar indicator variable is defined with slightly different semantics:

$$X_{ij} = \begin{cases} 1 & \text{IF } j \text{ IS AN SUCCESSOR OF } i, \\ 0 & otherwise \end{cases}$$

The analysis and computations to be performed afterwards will follow the same structure as before, producing in the same expectancy results for the predicate.

[1] Chernoff's bound

SECOND SOLUTION

Let's start with some key observations: the comparisons are just made with the chosen root k, any two elements are compared at most once, and every time a node is compared with the root k, it will increase its depth in the tree. Let denote with $n_1, \ldots, n_k$ the node of a BST (Binary Search Tree), where an $n_t \leq n_p \forall t \leq p$. Let's fix a generic node n_i then we have:

$$X_j = \begin{cases} 1 & \text{NODE } n_i \text{ IS A DESCENDENT OF } n_j \\ 0 & \text{OTHERWISE} \end{cases}$$

Therefore $X = \sum_{j=i}^{n} X_j$ is the hight (or the depth) of a generic node n_1. Therefore now we need to calculate the $E[X]$ (its expected value). Since the expected value is linear we have that $E[X] = \sum_{j=i}^{n} E[X_j]$, and since we know that $E[X_j] = P[X_j = 1]$ we should approximate the latter probability. Since a couple of element can be compared at most once and every comparison means a comparison with the root (an increasing of the depth), we can can assume that: if n_i is in the left(right) subtree of n_j it means that there are at most $j-i+1$ elements in the left(right) subtree. Since the subtree has $j-i+1$ elements, and because root are chosen randomly and independently, the probability that any given element is the first one chosen as a root is $\frac{1}{j-i+1}$.Therefore we have:

$$\begin{aligned} P(X_j = 1) =& P[n_i \text{ IS A DESCENDENT OF } n_j] \\ \leq& P[n_i \text{ IS IN THE LEFT SUBTREE } n_i \text{ IS IN THE RIGHT SUBTREE}] \\ =& \frac{1}{\text{NUMBER OF NODE IN THE LEFT SUBTREE}} + \frac{1}{\text{NUMBER OF NODE IN THE RIGHT SUBTREE}} \\ \leq& \frac{2}{j-i+1} \end{aligned}$$

Therefore we have $E[X] = \sum_{j=i}^{n} \frac{2}{j-i+1}$, if we change of variables[2] $k = j - i$ and we bound the harmonic series we have:

$$\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{k=1}^{n} \frac{2}{k} = 2ln(n)$$

Let's write down the Chernoff Bound:

Theorem 1 *(Chernoff Bounds). Let $X = \sum_{i=1}^{n} X_i$, where $X_i = 1$ with probability p_i and $X_i = 0$ with probability $1 - p_i$, and all X_i are independent. Let $\mu = E(X) = \sum_{i=1}^{n} p_i$[3]. Then*

$$\begin{aligned} &(i)\textbf{\textit{Upper Tail:}} P(X \geq (1+\delta)\mu) \leq e^{-\frac{\delta^2}{2+\delta}\mu} && \textit{for all } \delta > 0 \\ &(ii)\textbf{\textit{Lower Tail:}} P(X \leq (1-\delta)\mu) \leq e^{-\frac{\mu\delta^2}{2}} && \textit{for all } 0 < \delta < 1 \end{aligned}$$

Therefore we have:

$$\begin{aligned} P(X \geq (1+c)2ln(n)) \leq& e^{-\frac{c^2}{2+c}2ln(n)} \\ =& \frac{1}{e^{\frac{c^2}{2+c}2ln(n)}} \\ =& \frac{1}{n^{\frac{2c^2}{2+c}}} \\ =& \frac{1}{n^c} \text{WITH } c > 2 \end{aligned}$$

[2] LINK1

[3] Indicator variable

Karp-Rabin fingerprinting on strings

Given a string $S = S[0 \dots n-1]$, and two positions $0 \leq i < j \leq n-1$, the longest common extension $lce_S(i,j)$ is the length of the maximal run of matching characters from those positions, namely: if $S[i]! = S[j]$ then $lce_S(i,j) = 0$; otherwise, $lce_S(i,j) = max\{l \geq 1 : S[i \dots i+l-1] = S[j \dots j+l-1]\}$. For example, if $S = abracadabra$, then $lce_S(1,2) = 0$, $lce_S(0,3) = 1$, and $lce_S(0,7) = 4$. Given S in advance for preprocessing, build a data structure for S based on the Karp-Rabin fingerprinting, in O(n log n) time, so that it supports subsequent online queries of the following two types:

- $lce_S(i,j)$: it computes the longest common extension at positions i and j in O(log n) time.
- $equalS(i,j,l)$: it checks if $S[i \dots i+l-1] = S[j \dots j+l-1]$ in constant time.

Analyze the cost and the error probability. The space occupied by the data structure can be O(n log n) but it is possible to use O(n) space. [Note: in this exercise, a onetime preprocessing is performed, and then many online queries are to be answered on the fly.]

SOLUTION
Consider a string S of length n, we call $S[i]$ the i-th element in S starting from 0, and $S[i,l]$ the substring of S with length l starting at i (i.e. $S[i] \cdot S[i+1] \dots S[i+l-1]$, where $\cdot$ is the string concatenation). The idea is to build an array R with the same length of the string S such that for each index i we have that $R[i]$ contains the prefix of S of length $i+1$. $R[0]$ contains the first word of S, $R[1]$ contains the concatenation of the first two wolds of S etc. In general we will have that $R[i] = S[0, i+1]$. This representation has of course the problem to be too much expensive, but we will see later how to really implement the data structure, first have a look at how it works for checking the equality of two substring and for finding the lce_S of two indexes.

Given a pair of indexes i and j, and a length l we want to check whether $S[i,l] = S[j,l]$, looking at R. Considering that $\forall i, l. R[i+l] = S[0,i] \cdot S[i+1,l] = R[i] \cdot S[i+1,l]$, we can prove that

$$\forall i, l \,.\, S[i,l] = R[i+l-1] - R[i-1]$$

where we denote by $\alpha - \beta$ the string α without the prefix β (i.e. $\alpha\gamma - \alpha = \gamma$). Now we can simply check whether $s[i,l] = s[j,l]$ by comparing $R[i+l-1] - R[i-1]$ and $R[j+l-1] - R[j-1]$. The problem of finding the longest common extension is quite simple, it is sufficient to use some form of binary search, exploiting the array R. We use a recursive function in order to find the $lce_S(i,j)$, the input are i, j, the string S and its length n. We consider to have a function $is\_equal(A,x,y,l)$ such that return true if $A[x,l] = A[y,l]$. The main difference between this algorithm and a binary search is that we will continue calling recursively the function with a string of halved length regardless of the result of the equality check.

```
function LCE($S$,$i$,$j$,$n$)
    if $n = 1$ then
        if $is\_equal(S, i, j, l)$ then
            return $1$
        else
            return $0$
        end if
    end if
    $l = n/2$
    if $is\_equal(S, i, j, l)$ then
        return $l +$ LCE($S$,$i+l$,$j+l$,$n-l$)
    else
        return LCE($S$,$i$,$j$,$l$)
    end if
end function
```

Note that assuming the is_equal function to have constant cost we have that the cost of lce is $O(log(n))$ since at each iteration only a recursive call is reached and the length of the input is halved each time.

In the real implementation, we will use an array H which is the hash fingerprint of R. For each index i $H[i] = h(R[i]) = R[i] \; mod \; p$, of course we don't really built the array R, then we can simply fill $H[i]$ with $S[0,i+1] \; mod \; p$. An implicit assumption in this step (and in Karp-Rabin fingerprint in fact) is to

see each string as a number, in particular we see S as the representation of an integer number with base b (depending on the number of possible word in the alphabet). In the following we will call S the string and $(S)_b$ the number for which S is the representation in base b. We will also distinguish between $S[i]$ the word in position i of S and $S[i]_b$ the number between 0 and $b-1$ which represents. Note that we have to redefine h as $h(\alpha) = (\alpha)_b \; mod \, p$.
Then the number $(S)_b$ will be $S[0]_b \times b^{n-1} + S[1]_b \times b^{n-2} \cdots + S[n-1]_b \times b^0$. With this representation we have the beautiful feature that $(S[0, i+l])_b = (S[0, i] \cdot S[i, l])_b = (S[0, i])_b \times b^l + (S[i, l])_b$, that we can exploit to prove $h(S[i, l]) = H[i + l - 1] - H[i - 1] \times b^l \; mod \, p$.

$$\begin{aligned}
H[i + l - 1] = {} & h(R[i + l - 1]) \\
& h(S[0, i + l]) \\
& (S[0, i + l])_b \; mod \, p \\
& (S[0, i])_b \times b^l + (S[i, l])_b \; mod \, p \\
& ((S[0, i])_b \; mod \, p) \times b^l + ((S[i, l])_b \; mod \, p) \; mod \, p \\
& h(S[0, i]) \times b^l + h(S[i, l]) \; mod \, p \\
& h(R[i - 1]) \times b^l + h(S[i, l]) \; mod \, p \\
& H[i - 1] \times b^l + h(S[i, l]) \; mod \, p
\end{aligned}$$

This way the cost of compare two arbitrary substring of the same length $S[i, l]$ and $S[j, l]$ is the cost of acceding to four elements of H plus a constant number of arithmetic operations, i.e. $O(1)$. Since we have found a good approximation of the procedure is_equal, we can compute lce_S with the algorithm shown before in $O(log \; n)$ time.
The space occupied by the data structure is quasi $O(n)$ where n is the length of S. Actually the size of the input in bit is n times the size of the word (i.e. $n \times log \; b$) and the size of the output is n times the size of an element of H (i.e. $n \times log \; p$). Note that we can consider the use of R as a particular case in which p is equal to b^{n+1} (the maximum number that a string of length n with alphabet of size b can represent plus one), in this case the modulus in h is redundant and we have a data structure of size $O(n^2)$.
It is not difficult to see that we can use the compositional property of the h function in order to achieve a cost of $O(n)$ to fill H. The code speaks for itself probably, the idea is to compute $H[0]$ as $S[0] \; mod \, p$, $H[1]$ as $S[1] + S[0] \times b \; mod \, p$ etc. We use only a constant number of operations for each word in S.

```
$H[0] = S[0] mod p$
for $i \in \{1, \ldots n-1\}$ do
    $H[i] = H[i-1] \times b + S[i] \; mod \, p$
end for
```

We take p as a random prime number $p \in [2, \cdots, \tau]$, where $\tau > n$. Let's have a look at the collision cases: we have a collision when $S[i, l] \neq S[j, l]$ but $h(S[i, l]) = h(S[j, l])$. This is the same as

$$\begin{aligned}
& H[i + l - 1] - H[i - 1] \times b^l \; mod \, p = H[j + l - 1] - H[j - 1] \times b^l \; mod \, p \\
& H[i + l - 1] - H[i - 1] \times b^l - (H[j + l - 1] - H[j - 1] \times b^l) \; mod \, p = 0 \\
& h(S[i, l]) - h(S[j, l]) \; mod \, p = 0 \\
& ((S[i, l])_b \; mod \, p) - (S[j, l])_b \; mod \, p) \; mod \, p = 0 \\
& (S[i, l])_b - (S[j, l])_b \; mod \, p = 0 \\
& p \; divides \; (S[i, l])_b - (S[j, l])_b
\end{aligned}$$

We define $c = (S[i, l])_b - (S[j, l])_b$, we want to count how many bad choices we have for p (how many choices for p such that p divides c). Note that as $(S)_b$ also c is a number representable with a string of length n with alphabet of size b; then $0 \leq c \leq b^n$. Let k be the number of prime number dividing c, then $c = p_1^{i_1} \times p_2^{i_2} \times \cdots \times p_k^{i_k}$ and, since p_x is grater or equal than 2 and i_x is greater or equal than 1 for all x, we can say that $c \geq 2^k$ (as seen in class). Finally $2^k \leq c \leq b^n \leq b^{n \times log \; b}$, and $k \leq n \times log \; b$.Since the possible choices for a prime number in the interval $[2, \cdots, \tau]$ are approximately $\frac{\tau}{ln(\tau)}$, the probability of error is less or equal to $\frac{\#\text{BAD PRIMES}}{\#\text{PRIMES}} = \frac{n \times log \; b}{\frac{\tau}{ln(\tau)}}$. If we choose $\tau \approx n^{a+1} ln(n)$ then we have that the probability of error is less or equal than $\frac{log \; b}{n^a}$.

SOLUTION in O(N LOG N) works for every kind of hash function

The proposed data structure that maintain a series of trees. At each level we encode power of two elements (THIS IS IMPOSSIBLE TO EXPLAIN... LOOK THE CODE). The space occupied by this data structure is $O(nlog(n))$ where n is the length of the input array.

```
function CreateTree($S$,$n$,$p$)
    $A \leftarrow NEW\ Array[log_2(n) + 1]$
    $TEMP \leftarrow NEW\ Array[n]$
    for $i\ IN\ (0, n)$ do
        $TEMP[i] \leftarrow S[i]\ mod\ p$
    end for
    $A[0] \leftarrow TEMP$
    for $i\ IN\ (1, log_2(n))$ do
        $TEMP \leftarrow NEW\ Array[n - i]$
        for $j\ IN\ (0, n - i)$ do
            $TEMP[j] \leftarrow A[i - 1][j] + A[i - 1][j + 2^{i-1}]\ mod\ p$
        end for
        $A[i] \leftarrow TEMP$
    end for
    return $A$
end function
```

Now to have $lce_S(i, j)$ we build the implement the following procedure, where $h = \lfloor log_2(n - j)\rfloor$

```
function LCE($i$,$j$,$h$)
    if $A[h][i] == A[h][j]$ then
        return $2^h$
    else
        if $h \neq 0$ then
            return 0
        else
            return LCE($i, j, h - 1$)+LCE($i + 2^{h-1}, j + 2^{h-1}, h - 1$)
        end if
    end if
end function
```

Notice that, this procedure work in $O(log(n))$ since the array is length is at most log(n) and we are doing two recursive call with an array one unit smaller each time. Finally to obtain $equalS(i, j, l)$ in cost $O(1)$ we simply check whether $A[\lceil log_2(l)\rceil)][i] == A[\lceil log_2(l)\rceil][j]$ is true.

Notice that all this algorithm work for arrays in which their length n is a power of two. If we have an array that is not of the latter length, we are doing the following: create the same data structure as before but the part of the array that is not in the tree, that it's at most long $2^{i+1} - 2^i - 1$ where $i = \lfloor log_2(n)\rfloor$, is store in simple array. In this case whether we need to check if $A[\lfloor log_2(n - j)\rfloor][i] == A[\lfloor log_2(n - j)\rfloor][j]$, i.e. the longest possible string, we need also to check, manually, whether there is a matching in the array. Last but not least, the calculation of the error probability is exactly the same to the one analysed during the course.

Hashing sets

Your company has a database $S \in U$ of keys. For this database, it uses a randomly chosen hash function h from a universal family H (as seen in class); it also keeps a bit vector B_S of m entries, initialized to zeroes, which are then set $B_S[h(k)] = 1$ for every $k \in S$ (note that collisions may happen). Unfortunately, the database S has been lost, thus only B_S and h are known, and the rest is no more accessible. Now, given $k \in U$, how can you establish if k was in S or not? What is the probability of error? [Note: you are not choosing k and S randomly as the they are both given... randomization here is in the choice of $h \in H$ performed when building B_S] Under the hypothesis that $m \geq c|S|$ for some $c > 1$, find the expected number of $1s$ in B_S under a uniform choice at random of $h \in H$.

SOLUTION

a) To check whether $k \in U$ belong to S, we simply check $B_S[h(k)] = 1$. The probability of error is equal to $P(error) = 1 - (1 - \frac{1}{m})^{|S|}$. This problem is similar to the "Birthday paradox" (i.e. fixed a day how many people are born on the same day). In word: the probability of a collision is $\frac{1}{m}$, thus the probability to do not have a collision is $(1 - \frac{1}{m})$. If we have $|S|$ key the probability do not have any collision is $(1 - \frac{1}{m})^{|S|}$. Therefore the probability to have a collision is: $1 - (1 - \frac{1}{m})^{|S|}$. That is, the probability that there is at least one collision within the key of S, that is the probability that $\exists j \in S : h(k) = h(j)$ but $j \neq k$ with $k \in U$.

b) To estimate the size of S, we first create an indicator variable $X = \sum_{i=0}^{m-1} X_i$ where

$$X_i = \begin{cases} 1 & \text{IF } B_S[i] = 1 \\ 0 & \text{OTHERWISE} \end{cases}$$

Than the expectation $E[X]$ represents the expected number of 1 in the B_s table. To calculate the expectation we need to estimate the $P(B_S[h(k)] = 1)$, that, by the point a), is $1 - (1 - \frac{1}{m})^{|S|}$. Since we do not know $|S|$ we use the hypothesis, that is $|S| \leq \frac{m}{c}$. Therefore we have:

$$P(B_S[h(k)] = 1) = 1 - (1 - \frac{1}{m})^{|S|} \leq 1 - (1 - \frac{1}{m})^{\frac{m}{c}}$$

Hence we have $E[X] = \sum_{i=0}^{m-1} 1 - (1 - \frac{1}{m})^{\frac{m}{c}} = m - m(1 - \frac{1}{m})^{\frac{m}{c}}$. Now we have got a bound for $|S|$: $E[X] \leq |S| \leq \frac{m}{c}$.

Family of uniform hash functions

The notion of pairwise independence says that, for any $x1 \neq x2$ and $c1, c2 \in Z_p$, we have that

$$\underset{h\in\mathcal{H}}{Pr}[h(x_1) = c_1 \wedge h(x_2) = c_2] = \underset{h\in\mathcal{H}}{Pr}[h(x_1) = c_1] \times \underset{h\in\mathcal{H}}{Pr}[h(x_2) = c_2]$$

In other words, the joint probability is the product of the two individual probabilities. Show that the family of hash functions $H = \{h_{ab}(x) = ((ax + b)\ mod\, p)\ mod\, m\ : a \in Z_p^*, b \in Z_p\}$ (seen in class) is "pairwise independent", where p is a sufficiently large prime number $(m + 1 \leq p \leq 2m)$.

SOLUTION

Each probability from the right hand side of the equation are equal to $\frac{1}{m}$, simply because there are m buckets chosen uniformly (Universal hash function). Therefore, in the right hand side we have a probability equal to $\frac{1}{m^2}$.

On the left side of the equation, we will first count all the possible solutions (x_1, x_2), and then we will count only the good solutions (that is, the solutions x_1 and x_2 that match, respectively, c_1 and c_2).

$h(x_1)$ and $h(x_2)$ can be rewrite with their expression, such as:

$$\left.\begin{array}{l} h(x_1) = (a \cdot x_1 + b \text{ mod } p) \text{ mod } m \\ h(x_2) = (a \cdot x_2 + b \text{ mod } p) \text{ mod } m \end{array}\right\} x_1 \neq x_2$$

Now, let's consider the inner modulo p expressions as $r = a \cdot x_1 + b$ mod p and $s = a \cdot x_2 + b$ mod p, since p is prime and there is a linear transformation). Thus we can rewrite the original expression as: $Pr[r \text{ mod } m = c_1 \wedge s \text{ mod } m = c_2]$. Since $a \in [1 \ldots p-1] = \mathbb{Z}_p^*$ and $b \in [0 \ldots p-1] = \mathbb{Z}_p$, and both a and b are fixed for r and s, then there are $p \cdot (p-1)$ possible different combination for x_1 and x_2 $(x_1 \neq x_2)$.

Now, we focus on the "good" solutions. When p is sufficiently larger than m, the number of x_i that will match a given value c_i are $\frac{p}{m}$, as we have seen in class. However, we need to take in account modular arithmetic: not every values of m is going to be a multiple of p. Adding this to the analysis, the number of matches are bound by considering the round-to-floor and round-to-ceil limits:

$$\left\lfloor \frac{p}{m} \right\rfloor \leq \text{\# OF } x_i \text{ MATCHING } c_i \leq \left\lceil \frac{p}{m} \right\rceil$$

Since we have both c_1 and c_2, we need to consider their joint probability (i.g. multiplication), and then we can be bound it by combining both pieces of the analysis (dividing "good" solutions by all possible combination), giving us an approximation to the expected $\frac{1}{m^2}$:

$$\frac{1}{p \cdot (p-1)} \left(\left\lfloor \frac{p}{m} \right\rfloor \right)^2 \leq Pr[r \text{ mod } m = c_1 \wedge s \text{ mod } m = c_2] \leq \frac{1}{p \cdot (p-1)} \left(\left\lceil \frac{p}{m} \right\rceil \right)^2$$

It must be noted that the $\frac{1}{m^2}$ result may not be fully reached since that all the possible values a are $p - 1$, which make it impossible to remove the $\frac{p}{p-1}$ term. The only possibility is when $a = 0$ that would not make any sense algorithmically speaking. Therefore the result can only be approximated.

Deterministic data streaming

Consider a stream of n items, where items can appear more than once in the stream. The problem is to find the most frequently appearing item in the stream (where ties broken arbitrarily if more than one item satisfies the latter). Suppose that only k items can be stored, one item per memory cell, where the available storage is $k + O(1)$ memory cells. Show that the problem cannot be solved deterministically under the following rules: the algorithm can access only $O(log^c n)$ bits for each of the k items that it can store, and can read the next item of the stream; you, the adversary, have access to all the stream, and the content of the k items stored by the algorithm, and can decide what is the next item that the algorithm reads (please note that you cannot change the past, namely, the items already read by the algorithm) . Hint: it is an adversarial argument based on the k items chosen by the hypothetical deterministic streaming algorithm, and the fact that there can be a tie on $> k$ items till the last minute.

Show that the problem cannot be solved deterministically under the following rules: the algorithm can only use b bits, and read the next item of the stream, one item at a time. You, the adversary, have access to all the stream, and the content of the b bits stored by the algorithm: you cannot change those b bits and the past, namely, the items already read by the algorithm, but you can change the future, namely, the next item to be read. Since the algorithm must be correct for any input, you can use any amount of streams to be fed to the algorithm and as many distinct items as you want. [Hint: it is an adversarial argument based on the fact that, for many streams, there can be a tie on the items.]

SOLUTION

Any deterministic algorithm can be described by a deterministic finite state machine, in which every state given an input move deterministically to another state. Suppose to have a streaming of n element, k memory cell to store the frequencies, and θ are the bit necessary to store the algorithm state. Therefore we can have 2^θ possible states.

Let's suppose to input $2^\theta + 1$ times the same element x to the algorithm. Then, it must be the case that the algorithm is going to be in the same state in two different moments (notice that the possible state are 2^θ). Let's call A that particular state, let's use A_i and A_j to denote the two different moments in which we arrive in state A during the execution of the algorithm (without loss of generality we can assume $i < j - 1$, we will later handle the $i = j - 1$ case).

Now let's input to the algorithm an element $y \neq x$ for $(i + 1)$-times, starting from both A_i and A_j. Then let's proof that the algorithm replies in two different ways, depending on whether it starts from A_i or A_j:

Case 1: when we start from A_i, we will end up in this situation:
(for simplicity, we'll use a JSON-like syntax for representing the counters)
Counters: $\{x : i, y : i + 1\} \implies$ the algorithm outputs y as the most frequent element.

Case 2: when we start from A_i, we will end up in this situation:
Counters: $\{x : j, y : i + 1\} \implies$ since $j > i + 1$, the algorithm outputs x.

We've seen how, starting from the same state A and with the same input, the algorithm produces different outputs; therefore it cannot be deterministic.

case $i = j - 1$

If i=j-1, then the algorithm contains the transition $(A_i, x) \to A_j = A_{i+1}$. This means that the algorithms remains in state A after receving x. In particular, $A_{i+2} = \delta(\delta(A_i, x), x) = \delta(A_i, x) = A_i$, and thus we can just take $j = i + 2$ and we fall in the previous case $i < j + 1$.

Special case of most frequent item in a stream

Suppose to have a stream of n items, so that one of them occurs $> n/2$ times in the stream. Also, the main memory is limited to keeping just two items and their counters, plus the knowledge of the value of n beforehand. Show how to find deterministically the most frequent item in this scenario. [Hint: since the problem cannot be solved deterministically if the most frequent item occurs $\leq n/2$ times, the fact that the frequency is $> n/2$ should be exploited.]

SOLUTION

This solution works with just one counter. We initialize the counter with zero, then we do the following:

- add the first element of the stream to the counter with value 1
- if the next element is the same of the one stored by the counter, then we increment the counter by 1.
- if the element is different, we decrease the counter by 1, but if the counter become zero then we substitute the counter item with the new element, and we set the counter to zero.

Now, we give a little proof of correctness. Suppose we $x_1, \ldots, x_n$ element in the stream, with their relative frequency $f_1, \ldots, f_n$, and suppose the element element with maximum frequency is x_{i^*}. Therefore by hypothesis we have:

$$\sum_{\forall i, i \neq i^*}^{n} f_i < \frac{n}{2} < f_{i^*}$$

Then by the following mathematical steps we have:

$$\begin{aligned} \sum_{\forall i, i \neq i^*}^{n} f_i &< f_{i^*} \\ \sum_{\forall i, i \neq i^*}^{n} f_i - f_{i^*} &< 0 \\ f_{i^*} - \sum_{\forall i, i \neq i^*}^{n} f_i &> 0 \end{aligned}$$

Notice that this inequality is true by hypothesis. Thus the element remaining in the counter must be the one with higher frequency.

Count-min sketch: extension to negative counters

Check the analysis seen in class, and discuss how to allow $F[i]$ to change by arbitrary values read in the stream. Namely, the stream is a sequence of pairs of elements, where the first element indicates the item i whose counter is to be changed, and the second element is the amount v of that change (v can vary in each pair). In this way, the operation on the counter becomes $F[i] = F[i] + v$, where the increment and decrement can be now seen as $(i, 1)$ and $(i, -1)$.

SOLUTION

The frequency of the item i is represented by $\hat{F}[i] \to T[j, h_j(i)] = F[i] + X_{ji}$, where X_{ji} represent the garbage introduced by the other counters. If we just increment the counter X_{ji} is going to be positive, and then we can take the $min_j T[j, h_j(i)]$ to approximate $F[i]$. Instead, if we have also decrement, it could happen that $X_{ij} < 0$ therefore the method for the min is not going to work. In this case, we consider the absolute value of X_{ji}(i.e. $|X_{ji}|$) and we use $median_j T[j, h_j(i)]$ to approximate $F[i]$. Now let's proof that, with probability $1 - \delta^{1/4}$ holds:

$$F[i] - 3\epsilon\|F\| \leq \hat{F}[i] \leq F[i] + 3\epsilon\|F\|$$

First, let's do some consideration. The value of $\hat{F}[i] = median_j T[j, h_j(i)]$, and $T[j, h_j(i)] = F[i] + |X_{ji}|$. The first inequality(i.e. $F[i] \leq \hat{F}[i]$) holds because we took the absolute value of X_{ji}. Now we shall prove that $Pr[\hat{F}[i] > F[i] + 3\epsilon\|F\|] \leq 1/8$ [**NB: double check this last statement** (I'm not sure whether it is indeed $1/8$ or $\delta^{1/4}$)]. Taken j such that $\hat{F}[i] = median_j T[j, h_j(i)] = F[i] + |X_{ji}|$, then we have:

$$Pr[\hat{F}[i] \leq F[i] + 3\epsilon\|F\|]$$
$$Pr[F[i] + |X_{ji}| \leq F[i] + 3\epsilon\|F\|]$$
$$Pr[|X_{ji}| \leq 3\epsilon\|F\|]$$

From what we have seen in class and by the property of the absolute value, we have $E[|X_{ji}|] \leq E[X_{ji}] = \frac{\epsilon}{e}\|F\|$. Then we can apply the Markov inequality and since universal hash functions are pairwise independent, we have:

$$Pr[|X_{ji}| > 3\epsilon\|F\|] < \frac{E[|X_{ji}|]}{3\epsilon\|F\|} \leq \frac{\frac{\epsilon}{e}\|F\|}{3\epsilon\|F\|} = \frac{1}{3e} < \frac{1}{8}$$

Let's now define the indicator variable $Y = \sum_{j=0}^{r} Y_j$ which tell us the number of elements i (columns of the sketch) that have a garbage $|X_{ji}|$ greater than $3\epsilon\|F\|$.

$$Y_j = \begin{cases} 1 & \text{IF } |X_{ji}| > 3\epsilon\|F\| \text{ WITH } p < \frac{1}{8} \\ 0 & \text{OTHERWISE} \end{cases}$$

The median of $\hat{F}[i]$ is going to be a good approximation if we haven't got more than $\frac{r}{2}$ rows such that $|X_{ji}| > 3\epsilon\|F\|$ (that is we want $Y < \frac{r}{2}$).
Therefore, to calculate the probability of error, we calculate the probability of $Pr[Y \geq \frac{t}{2}]$. Here, we can use the Chernoff's Bound With: $(1 + \lambda)\mu = \frac{t}{2}$, $\mu = E[Y] = rp$.

$$\Pr[Y \geq (1+\lambda)\mu] < \left[\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}}\right]^{\mu} = \left[\frac{e}{e}\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}}\right]^{\mu} = \frac{1}{e^{\mu}}\left[\frac{e}{(1+\lambda)}\right]^{(1+\lambda)\mu} = \frac{1}{e^{rp}}\left[\frac{1}{(1+\lambda)}e\right]^{\frac{r}{2}} = \frac{1}{e^{rp}}[2pe]^{\frac{r}{2}}$$

Now we need to prove that $\frac{1}{e^{rp}}[2pe]^{\frac{r}{2}} \leq \delta^{\frac{1}{4}} = \frac{1}{2^{\frac{r}{4}}}$. If we use the reciprocal we have:

$$2^{\frac{r}{4}} \leq \frac{e^{rp}}{[2pe]^{\frac{r}{2}}} \leq \frac{1}{[2pe]^{\frac{r}{2}}}$$
$$2^{\frac{1}{4}} \leq \frac{1}{\sqrt{2pe}}$$

Since $e^{rp} \geq 1$. Then, we take $\frac{1}{2pe} > \sqrt{2}$, that is possible just if $p < \frac{1}{2\sqrt{2}e}$, indeed $p = \frac{1}{8}$.

Count-min sketch: range queries

Show and analyse the application of count-min sketch to range queries (i, j) for computing $\sum_{k=i}^{j} F[k]$. Hint: reduce the latter query to the estimate of just $t \leq 2\ log\ n$ counters $c_1, c_2, ..., c_t$. Note that in order to obtain a probability at most δ of error (i.e. that $\sum_{l=1}^{t} c_l > \sum_{k=i}^{j} F[k] + 2\epsilon\ log\ n||F||$), it does not suffices to say that it is at most δ the probability of error of each counter c_l: while each counter is still the actual wanted value plus the residual as before, it is better to consider the sum V of these t wanted values and the sum X of these residuals, and apply Markov's inequality to V and X rather than on the individual counters.

SOLUTION

We use the Dyadic Interval: a dyadic interval is a range of the form $[x2^y + 1...(x + 1)2^y]$ for parameters x and y. Each point in the range $[1 \dots n]$ is a member of $log_2 n$ dyadic interval, one for each y in the range $0 \dots log_2(n) - 1$ (think it as tree, where we split the interval in 2 at each level).

A count-min sketch table is kept for each set of dyadic ranges of length 2^y, one for each level in the tree, thus we have $log_2 n$ vectors $\widetilde{F_i}$.

Witnessing a new element in the stream will therefore trigger an update to all the $log_2 n$ tables.

The idea for a range query is to partition the range into dyadic intervals and return as result the sum of the values stored in the CMS tables for the corresponding intervals [see this link for further reference].

It can be shown[1] that any range will be split at most into $2log_2 n$ dyadic intervals.

Therefore for each query we access $t \leq 2\ log\ n$ counters $c_1, c_2, ..., c_t$. We then, want to proof $Pr[\sum_{l=1}^{t} c_l > \sum_{k=i}^{j} F[k] + 2\epsilon\ log\ n||F||] < \delta$.

Firstly, we notice that each counter c_i represents an interval $[a, b]$ and it value is $\sum_{k=a}^{b}(F[k]) + X_i$, where X_i is the rubbish of that particular interval. Then, give an interval $[l, r]$, and the counters we have $\sum_{i=1}^{t} c_i = \sum_{k=l}^{r} F[k] + X$, where X is the total error (notice that this works because the interval are disjoint). Then we substitute to the previous equation and we have:

$$Pr[\sum_{l=1}^{t} c_l > \sum_{k=l}^{r} F[k] + 2\epsilon\ log\ n||F||] < \delta$$

$$Pr[\sum_{k=l}^{r} F[k] + X > \sum_{k=l}^{r} F[k] + 2\epsilon\ log\ n||F||] < \delta$$

$$Pr[X > 2\epsilon\ log\ n||F||] < \delta$$

Now, we apply Markov inequality and by the linearity of the expectation we have:

$$Pr[X > 2\epsilon\ log\ n||F||] \leq \frac{E[X]}{2\epsilon\ log\ n||F||} \leq \frac{\sum_{i}^{2log_2(n)} E[X_i]}{2\epsilon\ log\ n||F||}$$

Now, $\forall i.E[X_i] < \frac{\epsilon}{e}||F_i||_1$, but $\forall i, j.||F_i||_1 = ||F_j||_1 = ||F||_1 \implies E[X_i] < \frac{\epsilon}{e}||F||_1$.
(by definition $||A||_1 = \sum_{h=1}^{n} |A[h]|$, that is in our case we sum the frequencies of symbols, which sum up to the same total amount no matter how we group them)

Thus we have

$$\frac{\sum_{i}^{2log_2(n)} E[X_i]}{2\epsilon\ log\ n||F||} \leq \frac{2\frac{\epsilon}{e}log\ n||F||}{2\epsilon\ log\ n||F||} = \frac{1}{e}$$

[1] First, note that:

- No three intervals of the same length can be contained in the partition of the same query, otherwise you could merge two of them;
- If there are two consecutive intervals of the same length, they must belong to two different larger "parent" intervals (otherwise you could replace them with their parent), i.e. their union cannot belong to the dyadic partition.

Thus, there are at most 2 intervals of each possible size in the partition. As the sizes are $log_2 n$ in all, the partition is made up of at most $2log_2 n$ intervals.

This is the probability of error (that the sum of garbage is more the $2\epsilon \; log \; n||F||$) in row j (the min, potentially distinct for each table).

Since we choose the row of each table that minimize the sum of the counter (by definition), then there must be an error in all the row r. Thus we have $\frac{1}{e^r} = \delta$ since $r = ln(\frac{1}{\delta})$.

Space-efficient perfect hash

Consider the two-level perfect hash tables presented in [CLRS] and discussed in class. As already discussed, for a given set of n keys from the universe U, a random universal hash function $h : U \to [m]$ is employed where $m = n$, thus creating n buckets of size $n_j \geq 0$, where $\sum_{j=0}^{n-1} nj = n$. Each bucket j uses a random universal hash function $h_j : U \to [m]$ with $m = n_j^2$. Key x is thus stored in position $h_j(x)$ of the table for bucket j, where $j = h(x)$.

This problem asks to replace each such table by a bitvector of length $n = n_j^2$, initialized to all 0s, where key x is discarded and, in its place, a bit 1 is set in position $h_j(x)$ (a similar thing was proposed in Problem 4 and thus we can have a one-side error). Design a space-efficient implementation of this variation of perfect hash, using a couple of tips. First, it can be convenient to represent the value of the table size in unary (i.e., x zeroes followed by one for size x, so 000001 represents x = 5 and 1 represents x = 0). Second, it can be useful to employ a rank-select data structure that, given any bit vector B of b bits, uses additional o(b) bits to support in O(1) time the following operations on B:

- $rank_1(i)$: return the number of 1s appearing in the first i bits of B.
- $select_1(j)$: return the position i of the jth 1, if any, appearing in B (i.e. B[i] = 1 and $rank_1(i) = j$).

Operations $rank_0(i)$ and $select_0(j)$ can be defined in the same way as above. Also, note that o(b) stands for any asymptotic cost that is smaller than $\Theta(b)$ for $b \to \inf$.

SOLUTION

The solution uses five data structures:

`H` is the Header table, it substitute the first level hash table and is memorized using a rank-select data structure;

`T` is the bitvector in which we will sore the bits for the presence/absence of the elements, it plays the role of all the second level tables;

`A` is the table that store the a parameters for the hash functions of the second level;

`B` is the table that store the b parameters for the hash functions of the second level;

`P` is the table that store the p parameters for the hash functions of the second level.

Before continue we make some observations:

1. $\sum_{j=1}^{n} n_j = n$, where n_j is the number of elements in the bucket j (called B_j from now on), as seen in class since by definition n is the number of elements to be stored;

2. in each bucket B_j we use an universal hash function h_j with $m = n_j^2$, and we have already proved that it's perfect (no collision) with probability $\geq \frac{1}{2}$;

3. given that $E[\sum_{j=1}^{n} n_j^2] < 2n$ and so by *Markov's inequality* that $Pr[\sum_{j=1}^{n} n_j^2 \geq 4n] < 1/2$, we know that $\sum_{j=1}^{n} n_j^2 \leq 4n$ with probability greater or equal to 1/2;

4. p, the parameter of the family of hash functions for the first level of hashing is a prime number greater than n, so for the *Bertrand's postulate* we know that we can choose a $p < 2n$ and we need less than $log_2 2n = log_2 2 + log_2 n = 1 + log_2 n$ to store it;

5. we take a and b (the parameters of the hash function in the first level) from $\mathbb{Z}_p$ and, since we know that we can choose a $p < 2n$, we need less than $log_2 2n = 1 + log_2 n$ bits for each one of them;

6. for each bucket B_j we choose randomly the parameters a_j and b_j of the hash function from $\mathbb{Z}_{p_j}$, with $p_j > n_j^2$ prime number (greater than the size of the bitvector), since we can choose $p_j < 2n_j^2$, the number of bits needed for each one of the two is less then $log_2 2{n_j}^2 = 1 + log_2 {n_j}^2$, the same number of bits are needed also for storing p_j itself.

Now we can speak more precisely about H and T.

H is simply the concatenation of the unary representation of n_1^2, n_2^2 ... n_n^2, so essentially n_1^2 times 0, followed by a 1 bit, then n_2^2 times 0, followed by a 1 bit, etc. We will use an additional amount of bits because of the rank-select data structure.

The table T is the concatenation of the bit vector of each bucket: the one for B_1 followed by the one for B_2, and so on, until B_n. Notice that the length of those buckets (and respectively of the vectors) can be different one from each other. For each bucket B_j we will call its bitvector Bv_j, we recall that the size of B_j is n_j and the size of Bv_j is n_j^2. We have

$$\begin{aligned} H &= (n_1^2)_1 \,|(n_2^2)_1 \mid \ldots \mid (n_n^2)_1 \\ T &= Bv_1 \,|Bv_2 \mid \ldots \mid Bv_n \end{aligned}$$

where $(\_)_1$ means the unary notation.

So we have that initially T is the concatenation of n_1^2 bits at 0, n_2^2 bits at 0 etc... Notice that in H we can consider the 1 bits as separator between the information about one bucket and the others, but we don't have anything similar in T: since we have concatenate all the bitvectors, we need some way to retrieve the range of any bitvector. To do so, we use the length of the bitvectors, contained in the header H.

The procedure for setting to 1 the bits relative to a given key x, and for looking whether the key is in S or not is the following:

$$\begin{aligned} j &= h_{abpn}(x) \\ k &= H.select_1(j-1) \;\#\; \text{POSITION OF THE } j-1_{th} \; 1 \\ \texttt{base} &= H.rank_0(k) \;\#\; \text{NUMBER OF 0S BEFORE THE INDEX } k \\ a' &= A[j] \\ b' &= B[j] \\ p' &= P[j] \\ m' &= H.select_1(j) - H.select_1(j-1) - 1 \\ \texttt{offset} &= h_{a'b'p'm'}(x) \\ T[\texttt{base} + \texttt{offset}] &= 1 \; // \; return \; T[\texttt{base} + \texttt{offset}] \end{aligned}$$

where a and b are the parameters of the hash function in the first level, and the last operation is $T[\texttt{base} + \texttt{offset}] = 1$ if we are filling the table (creation phase), and $return\ T[\texttt{base} + \texttt{offset}]$ if we are querying the hash table for key x. Notice that by definition of H, m' is the number of 0s between the $j-1_{th}$ and the j_{th} 1 and so it is n_j^2. Notice also that we can calculate base as $k-j$ since all but j bits are at 0 from the position 0 to the position k. We used the notation $h_{abpm}(x)$ for $((ax+b)\ mod\ p)\ mod\ m$.

If we are querying the table and the value returned is 0 then the element is not present, instead if it is 1 we say that the element is present, with a probability of error (one side error as usual).

The space occupied by the data structures used here is the following:

H : the number of $0s = \sum_{j=1}^{n} n_j^2$ plus the number of $1s = \#\text{BUCKETS} = n$ plus the extra space for the rank-select data structure, so totally $\sum_{j=1}^{n} n_j^2 + n + o(\sum_{j=1}^{n} n_j^2 + n) \leq 4n + n + o(4n+n) = 5n + o(n)$ with probability grater or equal 1/2;

T : the same number of $0s$ in H, $\sum_{j=1}^{n} n_j^2 \leq 4n$ whit probability grater or equal to $1/2$;

A : one array of n elements, each one needs less than $1 + log_2 {n_j}^2$ bits, then the total number of bits is less than $\sum_{j=1}^{n}(1 + log_2 {n_j}^2) = n + \sum_{j=1}^{n}(2 log_2 n_j) = n + 2\sum_{j=1}^{n}(log_2 n_j) < n + 2\sum_{j=1}^{n} n_j = n + 2n = 3n$;

B : by the same reasoning made for A we have less than $3n$ bits.

P : by the same reasoning made for A we have less than $3n$ bits.

Considering the space needed for a, b and p of the first level hash function, the total space used is less than: $5n + 4n + 3n + 3n + 3n + 3(1 + log_2 n) + o(n) = 18n + o(n)$.

Notice that if we use $p_j = p$ for each bucket B_j, than the size of A is $nlog_2 2n = n(1 + log_2 n) = n + nlog_2 n$, which is a problem since $nlog_2 n \notin o(n)$

Bloom filters vs. space-efficient perfect hash

Recall that classic Bloom filters use roughly $1.44log_2(1/f)$ bits per key, as seen in class (where $f = (1-p)^k$ is the failure probability minimized for $p \approx e^{-\frac{kn}{m}=1/2}$). The problem asks to extend the implementation required in Problem 10 by employing an additional random universal hash function $s : U \to [m]$ with $m = \lceil \frac{1}{f} \rceil$, called signature, so that s(x) is also stored (in place of x, which is discarded). The resulting space-efficient perfect hash table T has now a one-side error with failure probability of roughly f, as in Bloom filters: say why. Design a space-efficient efficient implementation of T, and compare the number of bits per key required by T with that required by Bloom filters.

SOLUTION

Now the data structure has got one-side error with probability because if we use the hash table s, which has got m elements, the probability of a collision is $\frac{1}{m}$. Since, $m = \frac{1}{f}$ we have: $Pr[error] = \frac{1}{\frac{1}{f}} = f$.

We consider first of all to have the same data structure of EX10. Therefore, we have H, T, A, B, P, and then till now if we have n keys we occupy a space equal to $18n + o(n)$ bits. Now we add another data structure T' where we store the hash generated by s. T' has got n slots, as the number of keys, in which every element occupy $log_2(m) = log_2(1/f)$ bits, since $s : U \to [m]$ with $m = \lceil \frac{1}{f} \rceil$. Now, since this data structure is static, ones we build the H, T, A, B, P we can insert each $s(k)$ in T' in the same position as it is in T. For example, the first element of bucket 1 (if it exist) is going to be the fist element of T', and so on. Then now to find the position of the hash in T' given a key k we use $rank_1(\textsc{base} + \textsc{offset})$, where $\textsc{base} + \textsc{offset}$ is the position calculate to check the if there is a 1 in T ($T[\textsc{base} + \textsc{offset}]$, look EX10).

Now we need to check how many bits are used for each key. T' uses $log_2(1/f)$ bit per key as explained before, and, by the previous analysis, we have 18 bit for each key + o(1)($18n + o(n)$). Therefore, we have roughly $log_2(1/f) + 18$ per key, instead Bloom filters use roughly $1.44log_2(1/f)$. Let's compare them, to see which is more convenient:

$$\begin{aligned} log_2(1/f) + 18 &< 1.44log_2(1/f) \\ -log_2(f) + 18 &< -1.44log_2(f) \\ -log_2(f) + 1.44log_2(f) &< -18) \\ log_2(f) &< -\frac{18}{0.44}) \\ f &< \frac{1}{2^{\frac{18}{0.44}}} \end{aligned}$$

Then perfect hash is convenient for f less than $\approx 4.84 \times 10^{-13}$.

MinHash sketches

As discussed in class, for a min-wise independent family H, we can associate a sketch

$$s(X) = \langle min\ h_1(X), min\ h_2(X), \dots min\ h_k(X) \rangle$$

with each set X in the given data collection, where $h_1, h_2, \dots, h_k$ are independently chosen at random from H. Consider now any two sets A and B, with their sketches $s(A)$ and $s(B)$. Can you compute a sketch for $A \cup B$ using just $s(A)$ and $s(B)$ in $O(k)$ time? Can you prove that it is equivalent to compute $s(A \cup B)$ from scratch directly from $A \cup B$?

SOLUTION

We have the two sketches of A and B:

$$s(A) = \langle min\ h_1(A), min\ h_2(A), \dots min\ h_k(A) \rangle$$
$$s(B) = \langle min\ h_1(B), min\ h_2(B), \dots min\ h_k(B) \rangle$$

We want to compute the sketch for $A \cup B$:

$$s(A \cup B) = \langle min\ h_1(A \cup B), min\ h_2(A \cup B), \dots min\ h_k(A \cup B) \rangle$$

For each $i \in 1, 2, \dots k$ we will take $min\ \{\ min\ h_i(A),\ min\ h_i(B)\ \}$ as $min\ h_i(A \cup B)$. This procedure require a constant number of operation (exactly one min) and must be repeated k times, so the time needed is $O(k)$.

In order to prove that what we obtain is exactly the sketch of $A \cup B$ we exploit the following lemma.

Lemma 1. *For each $i \in \{1, 2, \dots k\}$ it holds $h_i(A \cup B) = h_i(A) \cup h_i(B)$.*

Proof. take any x, then all the following are equivalent:

$$\begin{aligned}
& x \in h_i(A \cup B) \\
& \exists y \in A \cup B\ .\ h_i(y) = x \\
& \exists y \in A\ .\ h_i(y) = x \quad OR\ \ \exists y \in B\ .\ h_i(y) = x \\
& x \in h_i(A) \qquad OR \qquad x \in h_i(B) \\
& x \in h_i(A) \cup h_i(B)
\end{aligned}$$

□

Finally.

Theorem 2. *For each $i \in \{1, 2, \dots k\}$ it holds $min\ h_i(A \cup B) = min\ \{\ min\ h_i(A),\ min\ h_i(B)\ \}$.*

Proof. take any $i \in \{1, 2, \dots k\}$, then

$$\begin{aligned}
& min\ h_i(A \cup B) = && [using\ lemma 1] \\
& min\ (h_i(A) \cup h_i(B)) = && [trivial\ property\ of\ min] \\
& min\ \{min\ h_i(A),\ min\ h_i(B)\}
\end{aligned}$$

□

Randomized min-cut algorithm

Consider the randomized min-cut algorithm discussed in class. We have seen that its probability of success is at least $1/\binom{n}{2}$, where n is the number of its vertices.

- Describe how to implement the algorithm when the graph is represented by adjacency lists, and analyze its running time. In particular, a contraction step can be done in $O(n)$ time.
- A weighted graph has a weight $w(e)$ on each edge e, which is a positive real number. The min-cut in this case is meant to be min-weighted cut, where the sum of the weights in the cut edges is minimum. Describe how to extend the algorithm to weighted graphs, and show that the probability of success is still $\geq 1/\binom{n}{2}$. [hint: define the weighted degree of a node]
- Show that running the algorithm multiple times independently at random, and taking the minimum among the min-cuts thus produced, the probability of success can be made at least $1 - 1/n^c$ or a constant $c > 0$ (hence, with high probability).

SOLUTION

First Point

We shall keep nodes in an array. We expect to run $N-2$ contractions, spawning a new node for each of them, thus the array shall be of length $2N-2$, the last $N-2$ elements initially NULL.

Nodes will be annotated with their *degree* and a *container* field with the ID of the node they're currently "contained" in (initially themselves, then the nodes they get contracted into).

An edge (i,j) shall be a vector of three elements $< dest, mult, reverse >$:

- *dest*: the node ID of the destination (i.e. j);
- *mult*: recall we're in a multigraph, so there can be many edges $(i,j)_k$. We represent them as a single edge annotated with a *multiplicity* value, i.e. $(i,j).mult = |\{(i,j)_k\}|$.

  This way, each adjacency list contains $O(N)$ elements;
- *reverse*: as we're in an undirected graph, for each edge (i,j) there is the reverse (j,i). In *reverse* we store a pointer to (j,i), as we'll need it for bookkeeping during contraction operations.

Adjacency lists shall be doubly linked and sorted on *dest* fields.

Random choice of edge

Let N be the initial number of nodes, n be the number of remaining unique nodes.

Let M be the initial number of edges, m be the number of remaining edges.

Let $\delta(i)$ be the degree of node i (NB: contracted nodes will have degree 0). Then:

```
function ChooseEdge
    $r \leftarrow rand(0, 2m-1)$                                  ▷ note $\sum_{i=1}^{n} \delta(i) = 2m$
    $i \leftarrow 0$
    while $nodes[i].degree < r$ do          ▷ note we'll never reach the NULL tail of $nodes[]$
        $r \leftarrow r - nodes[i].degree$
        $i \leftarrow i + 1$
    end while
    $j \leftarrow 0$
    while $nodes[i].adj[j].mult < r$ do
        $r \leftarrow r - nodes[i].adj[j].mult$
        $j \leftarrow j + 1$
    end while
    return $(i,\ j)$

end function
```

You can think of what we're doing as having concatenated the adjacency lists and then crawled through them to edge r. This would require $O(n^2)$ steps done naively, but saving the degree of the node (the length of the adjacency list) allows us to skip to the next node in one step (a sort of skip-pointer).

In fact, what we do is crawl the list of nodes ($O(n)$) and then one adjacency list ($O(n)$ again).

This procedures results in a random uniform choice of the edge. In fact, the chance that a specific (directed!) edge $(i, j)_k$ is chosen ends up being:

$$P_r[(i, j_k)] = \frac{\delta(i)}{2 \times m} \times \frac{mult(i, j)}{\delta(i)} \times \frac{1}{mult(i, j)} = \frac{1}{2 \times m}$$

where

$$\frac{\delta(i)}{2 \times m} = P_r[node\ i\ is\ chosen]$$

$$\frac{mult(i, j)}{\delta(i)} = P_r[\{(i, j)_k\}|\ node\ i]$$

$$\frac{1}{mult(i, j)} = P_r[(i, j)_k|\ \{(i, j)_k\}]$$

$\{(i, j)_k\}$ is the set of all edges going from i to j

$mult(i, j) = |\{(i, j)_k\}|$ is the multiplicity of edge (i,j)

Thus, the probability that an (undirected!) edge $(i, j)_k$ is chosen results

$$\frac{1}{2m} + \frac{1}{2m} = \frac{1}{m}$$

Contraction

General idea for contracting over edge (i, j):

- Spawn a new node that will represent i and j contracted;
- Update the *container* field of nodes "contained" in i or j;
- Crawl through i and j's adjacency lists: nodes pointed by one but not the other get appended to the new node's list; nodes pointed by both are also appended, but their multiplicity is the sum of the old egdes' multiplicity.

  This can be done in linear time w.r.t. the length of the adjacency lists ($O(N)$) because they're sorted, so we proceed as in a *merge* operation of a MergeSort.

  In all cases, we take advantage of $(i, j).reverse$ to update the adjacency list of the pointed node with the new node. As the lists are doubly linked and assuming a pointer to the last element is kept, this can be done in $O(1)$ time.

```
function CONTRACT(i, j)
    $nID$ ← position of next NULL element in the tail of $nodes[]$
    $nodes[n] \leftarrow new\ Node()$
    for $x$ in $nodes$ do
        if $x.container = i \lor x.container = j$ then
            $x.container \leftarrow nID$
        end if
    end for
    $it_i$ ← iterator over $nodes[i].adj$
    $it_j$ ← iterator over $nodes[j].adj$
    while $it_i.hasNext()$ and $it_j.hasNext()$ do
        if $it_i.dest = j$ then
            $it_i.next()$
        else if $it_j.dest = i$ then
```

```
            $it_j.next()$
        else if $it_i.dest = it_j.dest$ then
            /* create a pair of edges, one the reverse of the other */
            /* constructor takes, in order, $dest$, $mult$, $reverse$ */
            /* also passing $e$ to its own constructor is questionable but this is just pseudocode */
            $e \leftarrow new\ Edge(it_i.dest, it_i.mult + it_j.mult, new\ Edge(nID, it_i.mult + it_j.mult, e))$
            /* remove old edges from the destination's adjacency list */
            $it_i.reverse.remove()$
            $it_j.reverse.remove()$
            /* put reverse in destinations; note $nID$ is highest so it goes to the tail*/
            $nodes[e.dest].adj.append(e.reverse)$
            /* put $e$ at the tail of $nodes[nID].adj$ */
            $nodes[nID].adj.append(e)$
            $it_i.next()$
            $it_j.next()$
            /* increment $nodes[nID]$'s degree by $e.mult$ */
            $nodes[nID].degree \leftarrow nodes[nID].degree + e.mult$
        else if $it_i.dest < it_j.dest$ then
            $e \leftarrow new\ Edge(it_i.dest, it_i.mult, new\ Edge(nID, it_i.mult, e))$
            $it_i.reverse.remove()$
            $nodes[e.dest].adj.append(e.reverse)$
            $nodes[nID].adj.append(e)$
            $it_i.next()$
            $nodes[nID].degree \leftarrow nodes[nID].degree + e.mult$
        else                                                    ▷ $it_j.dest < it_i.dest$
            $e \leftarrow new\ Edge(it_j.dest, it_j.mult, new\ Edge(nID, it_j.mult, e))$
            $it_j.reverse.remove()$
            $nodes[e.dest].adj.append(e.reverse)$
            $nodes[nID].adj.append(e)$
            $it_j.next()$
            $nodes[nID].degree \leftarrow nodes[nID].degree + e.mult$
        end if
    end while
    Process leftovers from either $i$ or $j$'s adjacency lists, if there are any, in a similar fashion.
    $nodes[i].degree \leftarrow 0$        ▷ degrees of $i$ and $j$ are put to 0 for ChooseEdge's conveniency
    $nodes[j].degree \leftarrow 0$        ▷ we could also detach the adjacency lists if we liked, but it's not necessary
end function
```

Computing the cut

After running $N - 2$ contractions, you'll end up with two nodes hl_1 and hl_2 (highlanders) left over in the graph.

The size of the cut will be $hl_1.degree = hl_2.degree.$

The *container* field of the first N elements of *nodes* will be either hl_1 or hl_2.

The two classes of nodes will thus define the cut. Should you need the actual edges making up the cut, you'll crawl through the original graph and output edges connecting nodes of the two classes ($O(N^2)$).

Second Point

In the algorithm we change the way we choose an edge to contract: edges with high weight will have a greater probability to be chosen with respect to others with lower weight. We define $weight : E \longrightarrow \mathbb{R}$ as the function that associate the weight to a given edge. We will use the *weight* function also on subsets of E, intending for $weight(X)$ with $X \subseteq E$ the sum of the weight of all the edges in X.

$$weight(X) = \sum_{e \in X} weight(e) \qquad\qquad where\ X \subseteq E$$

As seen in class, the *min cut* is not unique, nevertheless the sum of weights in all *min_ cuts* is the same, than with a notation abuse we will call it $weight(min\_cut)$. For each edge $x \in E$ the probability to be choose for a contraction is given by its weights normalized with the total sum of the weights of all the edges of the graph.

$$Pr[extract\ x] = \frac{weight(x)}{weight(E)}$$

Now the probability of making an error, when extracting an edge at random, is the probability of extracting one of the "bad" edges, i.e. edges such that their contraction cause a variation in $weight(min\_cut)$. We name BAD the set of "bad" edges.

$$Pr[error] = \sum_{e \in BAD} Pr[extract\ e] = \sum_{e \in BAD} \frac{weight(e)}{weight(E)} = \frac{weight(BAD)}{weight(E)} \tag{1}$$

As usual bad edges are those belonging to every *min cut*, so the total weight of bad edges is less or equal to the weight of *min cut*.

$$weight(BAD) \leq weight(min\_cut) \tag{2}$$

Given a node v, we define $star(v)$ as the set of all the edges touching v. For the same reasons explained in class, for each node v the weight of *min cut* must be less or equal to the sum of the weights of the edges touching v. This is because otherwise we would have a cut, $star(n)$, with weight less than *min cut*, which is absurd.

$$\forall\ v \in V\ .\ weight(min\_cut) \leq weight(star(v)) \tag{3}$$

We can reformulate the *handshaking lemma* for weighted graph in the following way, in which $weight(star(v))$ is something like the "weighted degree" of the node v.

$$weight(E) = \frac{\sum_{v \in V} weight(star(v))}{2} \tag{4}$$

Now we can derive:

$$\begin{aligned} weight(E) &= \frac{\sum_{v \in V} weight(star(v))}{2} && \text{(from 4)} \\ &\geq \frac{|V| weight(min\_cut)}{2} && \text{(from 3)} \end{aligned}$$

$$weight(min\_cut) \leq \frac{2\ weight(E)}{|V|} \tag{5}$$

Finally we have all the ingredients for the proof:

Theorem 1. *Selecting the edges for the contraction according to the probability described, each time we choose an edge the probability of choosing a "bad" edge is less or equal than $2/|V|$.*

$$Pr[error] \leq \frac{2}{|V|}$$

Proof.

$$\begin{aligned}
Pr[error] &= \frac{weight(BAD)}{weight(E)} && \text{(from 1)}\\
&\leq \frac{weight(min\_cut)}{weight(E)} && \text{(from 2)}\\
&\leq \frac{\frac{2\ weight(E)}{|V|}}{weight(E)} && \text{(from 5)}\\
&= \frac{2\ weight(E)}{|V| weight(E)} = \frac{2}{|V|}
\end{aligned}$$

□

The probability of making an error when selecting an edge is the same as for the case of a normal graph seen in class, so the probability of success for the algorithm (i.e. the probability of choosing well all the times) is still $\geq 1/\binom{n}{2}$.

Third Point

We know that $P(n) \geq 1/\binom{n}{2}$ is the probability of choose well for N times. We have seen also that the probability of error is $\leq (1 - 1/\binom{n}{2})$. Then the probability of having an error each time in N repetitions of the algorithm is

$$leq(1 - \frac{1}{\binom{n}{2}})^N \approx e^{-\frac{2N}{n(n-1)}}$$

Given a c we want the probability of success to be $\geq 1 - 1/n^c$. This is equivalent to ask that the probability of error is $\leq 1/n^c$. Making some calculations:

$$\begin{aligned}
\frac{1}{e^{\frac{2N}{n(n-1)}}} = e^{-\frac{2N}{n(n-1)}} &\leq \frac{1}{n^c}\\
n^c &\leq e^{\frac{2N}{n(n-1)}}\\
ln(n^c) &\leq ln(e^{\frac{2N}{n(n-1)}}) = \frac{2N}{n(n-1)}\\
N \geq \frac{n(n-1)ln(n^c)}{2}
\end{aligned}$$

So, for each given c, to make the probability of success at least $1 - 1/n^c$ is sufficient to take N grater or equal than $\frac{1}{2}n(n-1)ln(n^c)$.