HW Solution

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.0**

Submitted by:

- ≪**Alan Lee**≫: ≪**alanlee2**≫
- ≪**Joshua Burke**≫: ≪**joshuab3**≫
- ≪**Gerald Kozel**≫: ≪**gjkozel2**≫

(100 PTS.) Not a sorting question.

Consider an array $A[0 \,..\, n-1]$ with n distinct elements. Each element is an ℓ bit string representing a natural number between 0 and $2^\ell - 1$ for some $\ell > 0$. The only way to access any element of A is to use the function **FetchBit**(i, j) that returns the jth bit of $A[i]$ in $O(1)$ time.

1 (20 PTS.) Suppose $n = 2^\ell - 1$, i.e. exactly one of the ℓ-bit strings does not appear in A. Describe an algorithm to find the missing bit string in A using $\Theta(n \log n)$ calls to **FetchBit** without converting any of the strings to natural numbers.

2 (40 PTS.) Suppose $n = 2^\ell - 1$. Describe an algorithm to find the missing bit string in A using only $O(n)$ calls to **FetchBit**.

3 (40 PTS.) Suppose $n = 2^\ell - k$, i.e. exactly k of the $l-$bit strings do not appear in A. Describe an algorithm to find the k missing bit strings in A using only $O(n \log k)$ calls to **FetchBit**.

14 Solution:

1. Idea: Do a pass through the array checking the first bit with FetchBit. From there split the array values into two separate lists with index numbers. One array holds the indexes of the values that start with 0 and the other with 1. From there recurse until you reach an array that holds a singular value. At each recursal store the value of 1 or 0 associated with the array. Our base case is when you reach an array of size 1. At that point you kn

   Psuedocode: Page 1

   Analysis: Each pass will call FetchBit O(N) times, reducing by half at each level of the recursion tree. This is because we split the list into 2 at each point depending on we see a 1 or 0. Therefore the complexity of our calls to FetchBit would be:

   O(N)+O(N/2)+O(N/4)+O(N/8)+O(N/16)... = O(2N) = O(N)

   Now we have to consider the amount of levels we will create with this method. The function has to terminate with a level where one array has two indexes and the other has only one. This is similar to a full binary tree with a single node missing. We know that a full binary tree will have $1 + floor(log_2(n))$ levels, giving us $log_2(n)$ levels without counting the root of the tree. Therefore we do O(n) work log(n) times for each level, giving us a total of O(nlog(n)).

2. Idea: In the first pass go through the entire list and call FetchBit on the MSB of each value. instantiate two lists to store the indicies of each element we have checked depending on its MSB value; list of MSB one and list of MSB zero. Keep a count of 0's and 1's encountered through the loop. If the count of 0's is more than the count of 1's eliminate all values with MSB = 0 otherwise if the count of 1's is more than the count of 0's, eliminate all values with MSB = 1. While we are doing this, we must also add the index of each element to its respective list. Recurse through the next significant bit n times using the list of lesser size as the argument for the recursive call. Along the way return 0 or 1 at each step - 1 if the count of 0's is more and vice versa.

   Psuedocode: Page 2

   Analysis: On the first pass we use FetchBit n times. After that we have eliminated half of the values. Halfing the runtime continues until we arrive at an array with a single value. We can express this value as a geometric sum that converges to 1. This gives our final complexity of FetchBit as follows:

   O(N)+O(N/2)+O(N/4)+O(N/8)+O(N/16)... = O(2N) = O(N)

3. Idea: Follow similar method of finding missing values in 14.2. A difference is that now we have to handle the case where the counts are equal. In this case we have to recurse twice, one for the case where the count of 0's ends up being more and one for the case where the count of 1's is more.

   Psuedocode: Page 3

   Analysis: Run time follows from 14.2 with additional recursive step. In this case we have to make log(k) more calls to FetchBit as we have to split our recursion into two as we check both cases (missing 0 and missing 1). This depends on k because in the case where the counts are equal there are k more possible values missing from the remaining bits in the string. This gives a structure similar to a MergeSort where we recursively split our calls in two. Together this gives us a running time of O(nlog(k)).