Pset 4

Problem 0 - Suffix Trees on multiple strings (25%)

A suffix tree of m strings ($S_1, S_2, ..., S_m$) is a suffix tree built from inserting the strings in order: ($S_1\$_1$, $S_2\$_2$,...,$S_M\$_M$) where $\$_1$,...,$\$_M$ are distinct terminating strings.

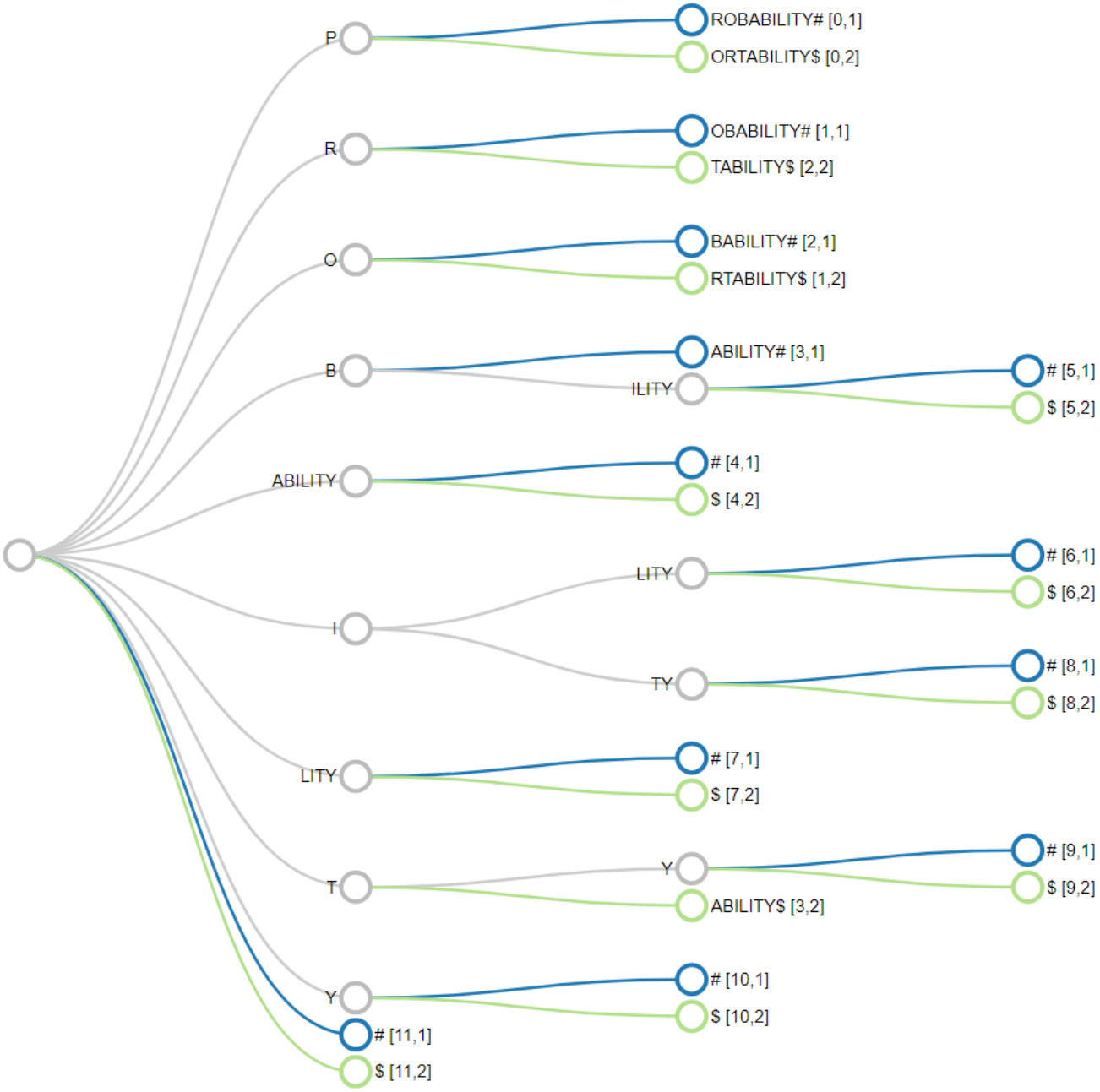
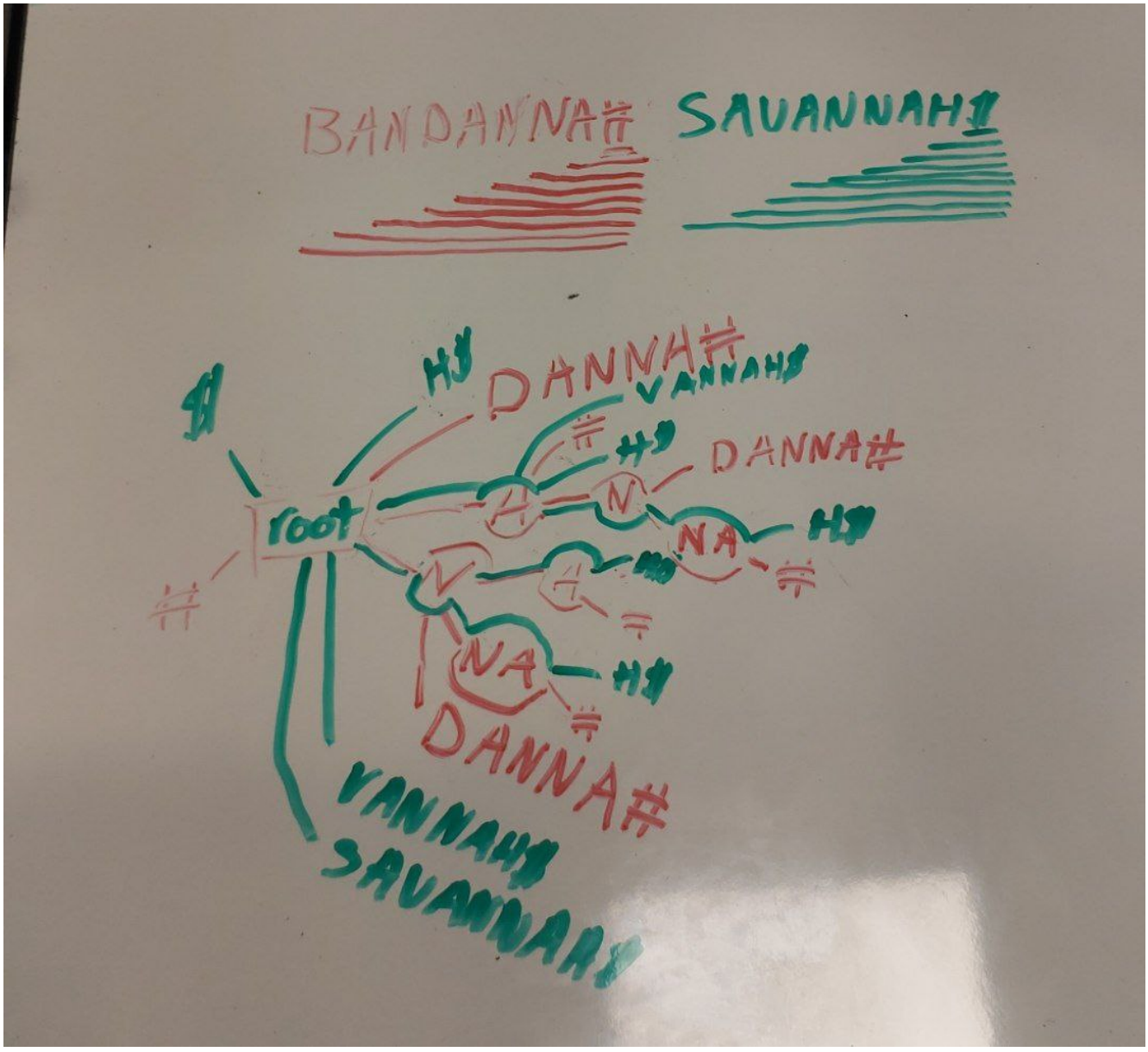


Figure 1: A suffix tree built for strings PROBABILITY# and PORTABILITY$ where # and $ are the terminating characters.

- Build the suffix tree for BANDANA# and SAVANAH$
- Give an algorithm to find the longest common substring of m strings, $S_1, ..., S_M$ using suffix trees.
- What is your algorithms runtime? Show your work/give an argument.

- Using your algorithm (by tracing the steps in your suffix tree), what is the longest common substring of BANDANNA# and SAVANNAH$?

Solutions:

1. 

2. To make an algorithm to find the longest common substring of m strings, we would first enter each node into the substring, and give each node a weight equal to the length of the substring (A would be weight 1, NA would be weight 2, etc.) to give us a baseline for length of the string. We then must find the max sum of strings in which both a $ and # is reachable from the node. The best way to do this would be to use depth first search keeping track of the length and only returning lengths in which a # and $ are eventual children of the node (on my graph, these are nodes which are encapsulated by both green and red). Return the maximum of these length values that meet this condition.
3. This algoritm's runtime is made up of three parts, building the tree, which costs n for every {s,...m} where n is the length of the substring. This gives us a runtime of this section being ($m * n$)Then we run depth first search, keeping track of the aforementioned variables which has a runtime of (nodes * Edges), and we know the maximum number of nodes in one string is n and the maximum number of strings (if all strings share no substrings) are m, making this value also $m * n$. The algorithm

would then return the LCS. Because we know these constituent parts would make up $2 * m * n$ then we know that the runtime simplifies to $m * n$.

4. The Longest common substring of BANDANNA# and SAVANNAH$ is ANNA, which is seen on my graph as the longest string encapuslated by both red and green.

Problem 1 - Reversible substrings (25%)

Let S be a string of length m and a substring be a contiguous sequence of characters from S identified by a tuple $S_{(i,l)}$ for starting index i and length l. Let the reverse of a string S[0],S[1],...,S[m-2],S[m-1] be S[m-1]S[m-2]...S[1]S[0] denoted S^{-}. A *reversible substring* is a substring where $S_{(i,l)} = S^{-}_{(m-(i+l),l)}$. Consider this example in python:

```
S = "PPABCDCBTTHMN"
i=3
l=5
m=len(S)

(S[::-1])[m-(i+l):m-(i+l)+l]
'BCDCB'
S[i:(i+l)]
'BCDCB'
```

Note that in the previous example we convert the length *l* into an ending index for python. Intuitively, a reversible substring is a substring that is the same in the forward and reverse directions.

- Write an algorithm using suffix trees that computes the longest reversible substring of a string *S*. *Hint: Think about using a suffix tree for multiple strings.*
- What is your algorithms runtime? Show your work/give an argument.
- Use your algorithm to find the longest reversible substring of the string RACECARS (you need not draw a diagram).

Solutions

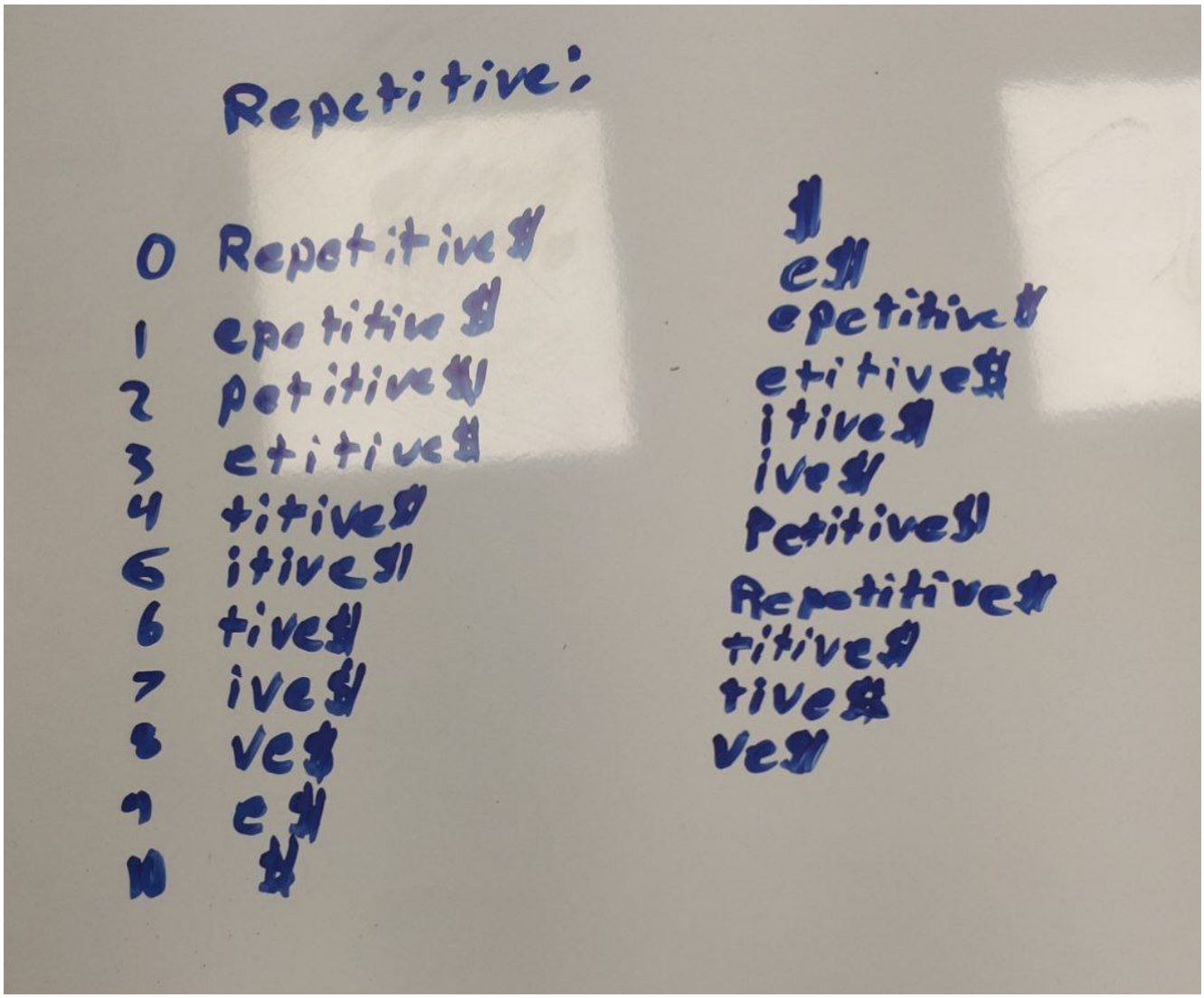
1. An algorithm using suffix trees that computes the longest reversible substring of a string would be using the above algorithm, but having the inputs be the string in the forward direction, and in the reverse direction. By then finding the longest common subsequence between the forward and reverse strings, we will have found the longest palindrome.
2. This runtime is the same as the above algorithm,
3. My algorithm found RACECAR.

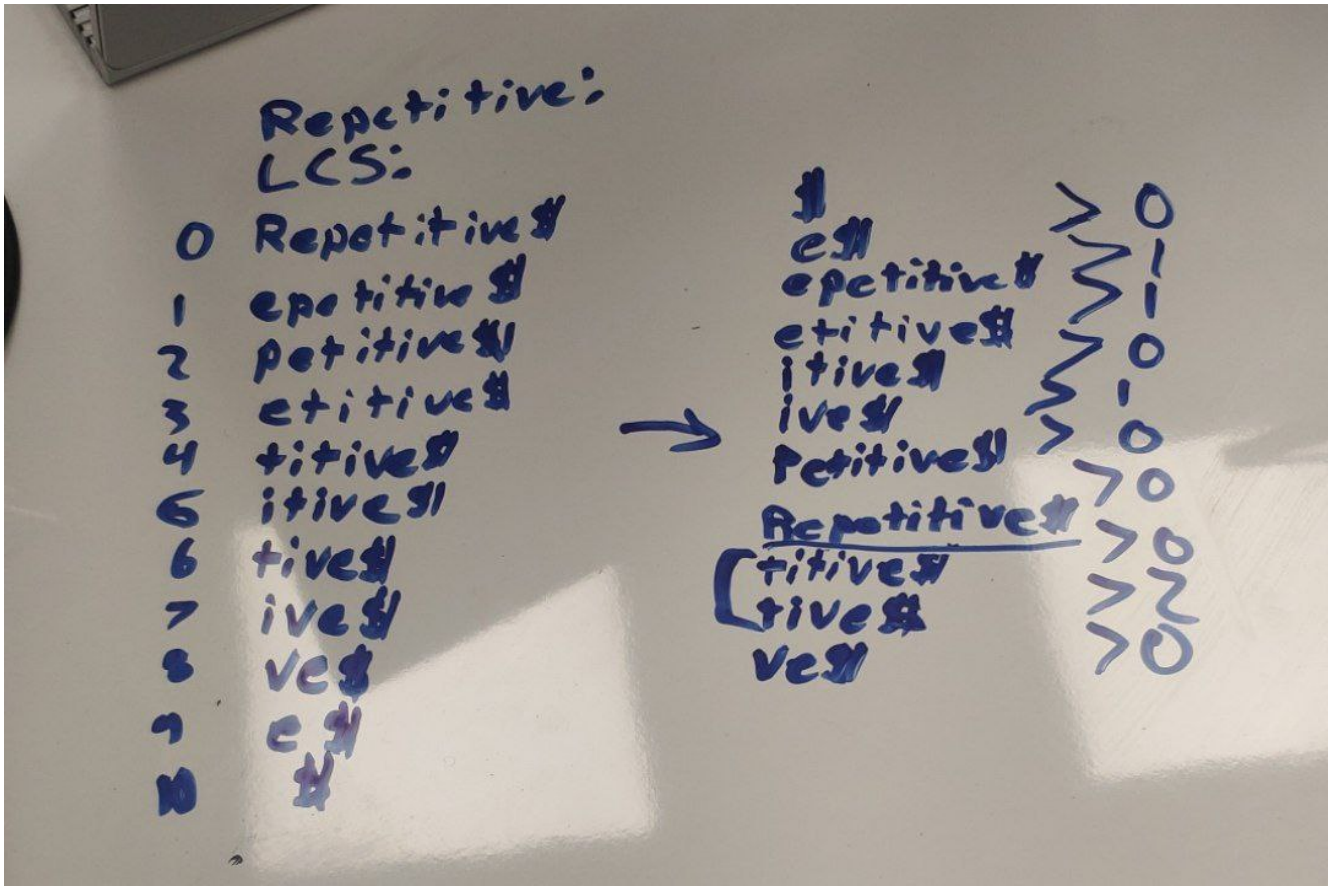
Problem 2 - Suffix Arrays (25%)

- Build the suffix array for string "repetititve".

- Recall the longest common prefix array, an array of size $m - 1$ that strores the length of the longest common prefix for each pair of adjacent strings. Build the LCP array for "repetitive".
- Describe an algorithm for finding all occurrences of a string P in a larger string S using suffix and LCP arrays built from S.
- Consider this algorithm and the case where the alphabet is large, say, *O(m)*. Would you prefer to use suffix trees or suffix arrays? Why?

Solutions

1. 

2. 

3. To find all occurences of a string prefix in a larger string S using the suffix, array, you must first build the suffix array. Then to find the occurences, since the array is sorted lexographically you can do a binary search, starting at the middle and testing if the value you are searching for is greater than, less than, or equal to the string you're comparing to. This will allow you to find the prefixes that interest you, then calculate the number of strings with prefixes (found by taking the number stored in the LCP array) or by counting the occurences of the prefix that is being searched for (this could be done by searching and counting the number of strings in which all the characters of p are found in $s_1 ... s_n$ as a prefix of s where the values LCP array is equivalent to the length of p) (in our previous example, there would be 2 where ti ws located because theres two occurences of ti found [if no occurences of the LCP array were larger than 1, ti could not exist in the array at all.]).
4. If you are working with a case where the alphabet is large, then it may make more sense to use a suffix array, because the array is sorted in lexographical order you can search it with binary search finding things in log(n) time before performing your checks. This means as the size of the library you're searching increases, the ammount of options you eliminate on each itteration of binary search on the strings will increase. This will cost more on the front end to build the array, but access will be much faster.

Problem 3 - Hashing (25%)

Consider the following simple hash function h for mapping keys $z_j, \in Z, h(z) = i$ with probability $\frac{1}{m}$ where $i = 1, ..., m$. What is the expected number of keys such that h(a) = h(b) over all keys a=/=b?
Hint: you want to define a similar random variable representing collisions as we discussed in class or Chapter 1.5 of the Algorithms textbook.

Solutions:

Knowns:
$z \in Z$
$|z| = n$
$h(z) = i$
$i = (1, ..., m)$
First, we know the probability of $h(z_1) =$ (any specific value) is $\frac{1}{m}$
Secondly, we know that the probability of $h(z_2) =$ (any specific value) is $\frac{1}{m}$
This means that the probability of $h(z_1) = h(z_2)$ is $\frac{1}{m^2}$
This hashing action happens m times meaning you have $m * \frac{1}{m^2} = \frac{1}{m}$
We know that the expected number of keys which will hash to the same value is the number of permutations where m choses 2 (m_1 maps to m_2) is $\binom{n}{2}$ where n is the number of values hashed.