Romanian Master of Informatics
6th Edition, Bucharest, 10th - 13th of October 2018

Password – solution

Author: Costin Oncescu

There are numerous approaches available; here is a handful of them.

Distinct symbols ($S + {}^{N(N-1)}/_{2}$ queries)

Since the symbols are distinct, $N \leq S \leq 26$. We can issue S queries each containing just one symbol, in order to determine which symbols appear in the password. Let the alphabet used be $a_1, a_2, \ldots, a_N$. Next, we issue (N choose 2) queries of the form $a_i a_j$ for all $i < j$. The answer can be 1, which means that a_j appears before a_i in the password, or 2, which means that a_i appears before a_j. This gives us a total order of the symbols; the password consists of the letters in this order.

"It doesn't get more exhaustive than this" ($O(N^2 \cdot S)$ queries)

Start with an empty string and insert symbols one by one. To perform the k-th insertion, try every symbol at every position in the existing string, until we get an answer of $k+1$.

To simplify the code, for every insertion start at the end and work backwards. This makes it easier to shift existing characters to the right one by one.

In theory this makes ${}^{N(N-1)}/_{2} \cdot S$ queries, but it performs better in practice.

"More X's... more X's... too many X's!" ($N \cdot S$ queries)

Insert all occurrences of one symbol before moving on to the next symbol. For a working string Q and a new symbol X, issue queries as follows: at every position in Q, insert one X, then 2 X's, then 3 X's, while the answer is equal to the new $|Q|$. When the answer stops increasing with $|Q|$, undo the last insertion and move on to the next position. Table 1 gives an example.

$N = 5, S = 3$, password = *bacca*		
a	$\to 1$	
aa	$\to 2$	
aaa	$\to 2$	$2 < 3$, so there are only 2 *a*'s.
baa	$\to 3$	
bbaa	$\to 1$	No values of 4 from now on, so there are no more *b*'s.
baba	$\to 2$	
baab	$\to 3$	
cbaa	$\to 1$	
bcaa	$\to 3$	
baca	$\to 4$	First value of 4 we encountered, so there is a *c* here.
bacca	$\to 5$	Terminates naturally once the last symbol is inserted.

Table 1: A possible interaction.

For a given X with frequency f_X, this algorithm will issue at most $|Q| + f_X + 1$ queries. Indeed, there will be f_X "correct" queries that guess a new occurrence and $|Q|+1$ "wrong" queries, one for each gap in Q.

At any given time, $|Q|$ is the sum of frequencies of previously explored symbols. It follows that the worst case is $N - S + 1$ occurrences of a followed by each of the other symbols once. In this case, the algorithm performs just under $N \cdot S$ queries, because $|Q|$ will be almost N the entire time we evaluate the last $S - 1$ symbols.

"XXXXXXXXX marks the spot" ($N \cdot S$ queries)

As before, insert all occurrences of one symbol before moving on to the next symbol. For a working string Q and a new symbol X, issue $|Q|+1$ queries as follows: take each prefix of Q and pad it with symbols X to length N. Taking differences between successive answers gives the number of X's between existing symbols in Q. Table 2 gives an example.

Again, the worst case is $N - S + 1$ occurrences of a and one each of the other symbols. This ensures that $|Q|$ grows as quickly as possible at the beginning. Working out the math yields a number slightly less than $N \cdot S$.

"Carry that weight a long time" ($N \cdot S/2$ queries)

The worst case of the previous solutions happens when they begin with a symbol with high frequency. Since $|Q|$ is the sum of frequencies of previously explored symbols, any frequent symbol we encounter early will carry towards the cost of all future symbols.

$N = 5, S = 3$, password = *bacca*		
aaaaa	$\to 2$	Now we know there are two a's.
aabbb	$\to 2$	2 is for *aa*, so there are no b's at the end.
abbbb	$\to 1$	1 is for a, so there are no b's here either.
bbbbb	$\to 1$	Now we know the string of all a's and b's is *baa*.
baacc	$\to 3$	3 is for *baa*, so there are no c's at the end.
baccc	$\to 4$	2 is for *ba*, so there must be two c's here.
bcccc	$\to 3$	
ccccc	$\to 2$	
bacca	$\to 5$	Print the correct password at the end.

Table 2: A possible interaction.

We can obviously alleviate this by processing symbols in increasing order of frequency. Start with the easier symbols and save the hard ones for the end. The worst case for this approach is when all frequencies are equal, $f = N/S$. Then the cost for the k-th distinct symbol is $1 + (k-1)f$. The total number of queries is:

$$\sum_{k=1}^{S}(1 + (k-1)f) = S + \frac{S(S-1)}{2}f = S + \frac{N(S-1)}{2}$$

To find the frequencies of each symbol, we need to run an additional $S - 1$ queries of the form $aaa\ldots a$, $bbb\ldots b$ and so on. The frequency of the last symbol can be found by subtracting the previous $S - 1$ values from N.

"Out of many, one" ($S + N \log_2 S$ queries)

This solution is inspired by the problem of merging k sorted arrays. As before, use $S - 1$ queries to find the frequencies of each symbol. Next, build S strings, one for each symbol: one containing all the a's, another containing all the b's and so on. Then, repeatedly, take the two shortest strings and **merge** them, until we are left with only one string.

Merging means that we start with two strings, each of which contains all the occurrences of a different set of symbols, in the same order as they appear in the password. We wish to obtain one string containing all the occurrences of the union of the two sets, again in the correct order. If we can do this, then after $S - 1$ merges there will be only one string, containing all the occurrences of every symbol in the correct order – that is, the password itself.

So how do we do this? Let us work out an example. Let our two strings be $A = abacaba$ and $B = defe$, containing all the occurrences of $a - c$ and $d - f$ respectively. Let the

strings left to merge		query	result	merged string so far
$abacaba$	$defe$	$adefe$	5	a
$bacaba$	$defe$	$abdefe$	6	ab
$acaba$	$defe$	$abadefe$	6	abd
$acaba$	efe	$abdaefe$	7	$abda$
$caba$	efe	$abdacefe$	8	$abdac$
aba	efe	$abdacaefe$	7	$abdace$
aba	fe	$abdaceafe$	9	$abdacea$
ba	fe	$abdaceabfe$	8	$abdaceaf$
ba	e	$abdaceafbe$	10	$abdaceafb$
a	e	$abdaceafbae$	11	$abdaceafba$
–	e	–	–	$abdaceafbae$

Table 3: Merging two strings.

password be $ab$$xxx$$dace$$yyyy$$afb$$zzz$$ae$ (for clarity, we have deemphasized symbols outside the $a - f$ range).

We begin by issuing the query $adefe$. If the answer is 5, then we know that the first a appears before d; otherwise d appears before the first a. In our case the answer is indeed 5, so we know that the merged string begins with a. Next we issue the query $abdefe$. The answer is 6, so ab must come before $defe$. The next query if $abadefe$. Here the answer is 3, not 7, so we know that aba does not come before $defe$; therefore, the first letters must be abd.

In general, queries will consist of (a) the string merged so far, (b) the first unmerged character of A and (c) the unmerged portion of B. If the answer is equal to the length of the query, we merge the next character from A. Otherwise, we merge the next character from B. Merging terminates when we reach the end of either string, after which we copy the remaining portion of the other string. Table 3 summarizes the entire run for this example.

Note that merging A and B makes at most $|A| + |B|$ queries, one per symbol merged. Suppose we simply merged $S/2$ pairs of strings regardless of their length. This would make at most N queries in total and cut the number of strings in half. Since there are $\log S$ iterations, this approach would make $N \log S$ queries in total. Merging the shortest strings performs no worse, but it can be much better in degenerate cases. For example, when the strings have length 1, 2, 4, 8, ..., our approach would make just $4N$ queries.

"Nothing left on the right side" ($S + N(1 + \log_2 S)$ queries)

This solution is based on the quickselect algorithm. Once again, use $S - 1$ queries to find the frequencies of each symbol. Now we know the composition of the password, but not the order. Choose a symbol at random (a "pivot"). We wish to

1. find the frequencies of symbols to the left and right of it;
2. solve the subproblem on the left side recursively;
3. output the pivot itself;
4. solve the subproblem on the right side recursively.

If the subproblem contains S' distinct symbols (S' will get smaller and smaller as we recurse into smaller subproblems), then finding the frequencies can be done in $S' - 1$ queries. We do not need to run a query for the pivot. If there are 10 copies of it and we randomly chose the 7th, then we automatically know that there are 6 more copies of it on the left and 3 on the right.

How do we build these queries? Assuming we chose the k-th occurrence of X, and if we wish to count the number of Y's to the left of X, then our query must consist of:

- all occurrences of X to the left of the current subproblem;
- k more occurrences of X;
- all the occurrences of Y *except* those to the left of the current subproblem (it is important not to include too many symbols or we may exceed N);

The answer will account for

- the number of X's to the left of the current subproblem;
- k more occurrences of X;
- the number of Y's right of the pivot in the current subproblem;
- the number of Y's to the right of the current subproblem.

If we choose our data structures carefully, we can compute the number of Y's left of the pivot by difference.

The base cases of this recursion are (a) no symbols left, when we simply return, and (b) a single symbol X with k occurrences, when we output k copies of X and return.

Computing the total effort is nontrivial, but we can get a good estimate as follows. For a subproblem of length N' containing S' distinct symbols, the expected number of queries performed is

$$Q(N') = S' - 1 + 2 \cdot Q(N'/2)$$

Let us split this recursion tree into a top part, where the subproblems have length $N' >= S$, and a bottom part, where $N' < S$. Let p be the first level in the bottom part (where the original problem has level 0). Then

$$p = \left\lceil \log_2 \frac{N}{S} \right\rceil$$

The top part has p levels and $1 + 2 + \cdots + 2^{p-1} < 2^p$ subproblems. Each of these subproblems is long enough that it can still have S distinct symbols (although in practice the numbers decrease quickly). Thus, the total work on these levels is at most

$$(S-1) \cdot 2^p = (S-1) \cdot 2^{\lceil \log_2 \frac{N}{S} \rceil} \approx (S-1)\frac{N}{S} < N$$

The bottom part starts with 2^p subproblems of length at most S. The next level has 2^{p+1} subproblems of length at most $S/2$, and so on. It follows that there are $\log_2 S$ levels. Each of these subproblems require effort proportional to their length (because there cannot be more distinct symbols than the length of the string). Thus, the total work on each level is $2^p \cdot S$ and the total work on the bottom part of the tree is at most

$$2^p \cdot S \log_2 S = 2^{\lceil \log_2 \frac{N}{S} \rceil} \cdot S \cdot \log_2 S \approx N \cdot \log_2 S$$

Adding up the initial frequency count and the work for the top and bottom parts of the tree, we get a total cost of $S + N(1 + \log_2 S)$ queries.