753. Cracking The Safe (/problems/cracking-the-safe/)

Dec. 24, 2017 | 4.8K views

There is a box protected by a password. The password is `n` digits, where each letter can be one of the first `k` digits `0, 1, ..., k-1`.

You can keep inputting the password, the password will automatically be matched against the last `n` digits entered.

For example, assuming the password is `"345"`, I can open it when I type `"012345"`, but I enter a total of 6 digits.

Please return any string of minimum length that is guaranteed to open the box after the entire string is inputted.

Example 1:

```
Input: n = 1, k = 2
Output: "01"
Note: "10" will be accepted too.
```

Example 2:

```
Input: n = 2, k = 2
Output: "00110"
Note: "01100", "10011", "11001" will be accepted too.
```

Note:

1. `n` will be in the range `[1, 4]`.
2. `k` will be in the range `[1, 10]`.
3. `k^n` will be at most `4096`.

Approach #1: Hierholzer's Algorithm [Accepted]

Intuition

We can think of this problem as the problem of finding an Euler path (a path visiting every edge exactly once) on the following graph: there are k^{n-1} nodes with each node having k edges.

For example, when `k = 4, n = 3`, the nodes are `'00', '01', '02', ..., '32', '33'` and each node has 4 edges `'0', '1', '2', '3'`. A node plus edge represents a *complete edge* and viewing that substring in our answer.

Any connected directed graph where all nodes have equal in-degree and out-degree has an Euler circuit (an Euler path ending where it started.) Because our graph is highly connected and symmetric, we should expect intuitively that taking any path greedily in some order will probably result in an Euler path.

This intuition is called Hierholzer's algorithm: whenever there is an Euler cycle, we can construct it greedily. The algorithm goes as follows:

- Starting from a vertex `u`, we walk through (unwalked) edges until we get stuck. Because the in-degrees and out-degrees of each node are equal, we can only get stuck at `u`, which forms a cycle.
- Now, for any node `v` we had visited that has unwalked edges, we start a new cycle from `v` with the same procedure as above, and then merge the cycles together to form a new cycle $u \to \cdots \to v \to \cdots \to v \to \cdots \to u$.

Algorithm

We will modify our standard depth-first search: instead of keeping track of nodes, we keep track of (complete) edges: `seen` records if an edge has been visited.

Also, we'll need to visit in a sort of "post-order", recording the answer after visiting the edge. This is to prevent getting stuck. For example, with `k = 2, n = 2`, we have the nodes `'0', '1'`. If we greedily visit complete edges `'00', '01', '10'`, we will be stuck at the node `'0'` prematurely. However, if we visit in post-order, we'll end up visiting `'00', '01', '11', '10'` correctly.

In general, during our Hierholzer walk, we will record the results of other subcycles first, before recording the main cycle we started from, just as in our first description of the algorithm. Technically, we are recording backwards, as we exit the nodes.

For example, we will walk (in the "original cycle") until we get stuck, then record the node as we exit. (Every edge walked is always marked immediately so that it can no longer be used.) Then in the penultimate node of our original cycle, we will do a Hierholzer walk and then record this node; then in the third-last node of our original cycle we will do a Hierholzer walk and then record this node, and so on.

Java | Python — Copy

```java
class Solution {
    Set<String> seen;
    StringBuilder ans;

    public String crackSafe(int n, int k) {
        if (n == 1 && k == 1) return "0";
        seen = new HashSet();
        ans = new StringBuilder();

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n-1; ++i)
            sb.append("0");
        String start = sb.toString();

        dfs(start, k);
        ans.append(start);
        return new String(ans);
    }

    public void dfs(String node, int k) {
        for (int x = 0; x < k; ++x) {
            String nei = node + x;
            if (!seen.contains(nei)) {
                seen.add(nei);
                dfs(nei.substring(1), k);
                ans.append(x);
            }
```

Complexity Analysis

- Time Complexity: $O(n * k^n)$. We visit every edge once in our depth-first search, and nodes take $O(n)$ space.
- Space Complexity: $O(n * k^n)$, the size of `seen`.

Approach #2: Inverse Burrows-Wheeler Transform [Accepted]

Explanation

If we are familiar with the theory of combinatorics on words, recall that a *Lyndon Word* `L` is a word that is the unique minimum of it's rotations.

One important mathematical result (due to Fredericksen and Maiorana (http://www-igm.univ-mlv.fr/~perrin/Recherche/Publications/Articles/debruijnRevised3.pdf)), is that the concatenation in lexicographic order of Lyndon words with length dividing `n`, forms a *de Bruijn* sequence: a sequence where every every word (from the k^n available) appears as a substring of length `n` (where we are allowed to wrap around.)

For example, when `n = 6, k = 2`, all the Lyndon words with length dividing `n` in lexicographic order are (spaces for convenience): `0 000001 000011 000101 000111 001 001011 001101 001111 01 010111 011 011111 1`. It turns out this is the smallest de Bruijn sequence.

We can use the *Inverse Burrows-Wheeler Transform* (IBWT) to generate these Lyndon words. Consider two sequences: `S` is the alphabet repeated k^{n-1} times: `S = 0123...0123...0123....`, and `S'` is the alphabet repeated k^{n-1} times for each letter: `S' = 00...0011...1122....` We can think of `S'` and `S` as defining a permutation, where the `j`-th occurrence of each letter of the alphabet in `S'` maps to the corresponding `j`-th occurrence in `S`. The cycles of this permutation turn out to be the corresponding smallest de Bruijn sequence (link (http://www.macs.hw.ac.uk/~markl/Higgins.pdf)).

Under this view, the permutation $S' \to S$ [mapping permutation indices $(i * k^{n-1} + q) \to (q * k + i)$] form the desired Lyndon words.

Java Python Copy

```java
class Solution {
    public String crackSafe(int n, int k) {
        int M = (int) Math.pow(k, n-1);
        int[] P = new int[M * k];
        for (int i = 0; i < k; ++i)
            for (int q = 0; q < M; ++q)
                P[i*M + q] = q*k + i;

        StringBuilder ans = new StringBuilder();
        for (int i = 0; i < M*k; ++i) {
            int j = i;
            while (P[j] >= 0) {
                ans.append(String.valueOf(j / M));
                int v = P[j];
                P[j] = -1;
                j = v;
            }
        }

        for (int i = 0; i < n-1; ++i)
            ans.append("0");
        return new String(ans);
    }
}
```

Complexity Analysis

- Time Complexity: $O(k^n)$. We loop through every possible substring.
- Space Complexity: $O(k^n)$, the size of `P` and `ans`.

Analysis written by: @awice (https://leetcode.com/awice).

Rate this article: **(/ratings/107/356/?return=/articles/cracking-the-safe/) (/ratings/107/356/?return=/articles/crackin**

Previous (/articles/ip-to-cidr/) Next (/articles/number-of-islands-ii/)

Join the conversation

Signed in as **tan7**.

Post a Reply

Z (https://discuss.leetcode.com/user/zhichao2)

zhichao2 commented 2 weeks ago

For the first solution,
if I change "Set<String> seen;" to "HashSet<String> seen;" I get the following error "
Exception in thread "main" java.lang.StackOverflowError
at java.util.HashMap.putVal(HashMap.java:598)
"
Anyone knows why?

J (https://discuss.leetcode.com/user/jso)

jso commented 3 months ago

@ryx (https://discuss.leetcode.com/uid/347817) for the second solution I think the reason the zeroes are appended is that the de Bruijin sequence allows substrings to wrap around but the safe cracking problem does not.

R (https://discuss.leetcode.com/user/ryx)

ryx commented 3 months ago

@awice (https://discuss.leetcode.com/uid/71269)
In the second approach, why do we need to append zeros at the end of the algorithm?
Thanks.

R (https://discuss.leetcode.com/user/ryx)

ryx commented 3 months ago

@chen806 (https://discuss.leetcode.com/uid/64010)
It is a Euler Circuit problem on an (n – 1)-dimensional de Bruijn graph.
So we need to cover all edges, not only all nodes.

X (https://discuss.leetcode.com/user/xllllx)

xllllx commented 3 months ago

Hierholzer's Algorithm goes every edge only once and there are k**n edges. (K**(n-1) nodes and each node has k out edges)

Shouldn't the time/space complexity be O(k**n) ?

C (https://discuss.leetcode.com/user/chen806)

chen806 commented 3 months ago

Is this a Euler Circuit problem?
We do not need to cover all edges but only all nodes.

D (https://discuss.leetcode.com/user/dcx9306)

dcx9306 commented 3 months ago

@awice (https://discuss.leetcode.com/uid/71269) Thanks awice! I did write it down. But I don't think it's just simply S'-->S mapping as mentioned in the article. I can't figure out why it's a valid Bruijin sequence. Is there any proof for the soundness? I appreciate it!

N (https://discuss.leetcode.com/user/njucs-cxgmail-com)

njucs.cxgmail.com commented 3 months ago

The first solution is really confused. I think it is equal to start from 00..0(k-1) using decrease value(eg. k-1, k-2, ... 0) preorder.


(https://discuss.leetcode.com/user/awice)

awice commented 3 months ago

@dcx9306 (https://discuss.leetcode.com/uid/300526) Basically we want the cycles of some permutation. The while loop is just finding a cycle and recording the cycle as we proceed. I will explain this better during the rewrite, but basically imagine the permutation is like [0, 3, 1, 2, 4]. We will record the cycles in order: (0) (1 3 2) (4).

D

dcx9306 commented 3 months ago

(https://discuss.leetcode.com/user/dcx9306)

Could you please clarify the while loop in the second solution? How does it produce a valid Bruijn sequence?

View original thread (https://discuss.leetcode.com/topic/114849)

Load more comments...