

A PROJECT REPORT

TOPIC : CRACKING THE SAFE

Submitted by

RITHIGA B M

192210205

Under the guidance of

Dr. A. GNANA SOUNDARI

in partial fulfilment for the completion of course

CSA0697-Design and Analysis of Algorithms for Amortized Analysis



**SIMATS ENGINEERING
THANDALAM
SEPTEMBER 2024**

ABSTRACT:

This paper presents a computational solution to the problem of unlocking a safe that is secured by a password consisting of a sequence of digits. Each digit in the password is selected from a range $[0, k-1]$. The unique characteristic of this safe is that it checks the most recent n digits entered each time a new digit is typed. The challenge is to generate the shortest possible string such that every potential password of length n appears as a substring within that string. This problem can be solved efficiently using De Bruijn sequences, which ensure that all possible combinations of digits of a specific length are contained exactly once in the sequence. The provided C program facilitates the generation of this sequence, accepting user-defined values for n (the length of the password) and k (the range of digits). Through this method, a practical approach is demonstrated for solving the problem efficiently by generating the required sequence and displaying the result.

PROBLEM STATEMENT AND ASSUMPTIONS:

The scenario revolves around a safe that is protected by a password composed of n digits, each selected from a range $[0, k-1]$. The safe has an unusual password verification process: instead of checking the entire input at once, it verifies only the most recent n digits entered after each keystroke. For instance, if the correct password is "345" and the user inputs "012345," the safe will verify the most recent 3 digits after each keystroke:

- After typing "0," the most recent 3 digits are "0," which is incorrect.
- After typing "1," the most recent 3 digits are "01," which is incorrect.
- After typing "2," the most recent 3 digits are "012," which is incorrect.
- After typing "3," the most recent 3 digits are "123," which is incorrect.
- After typing "4," the most recent 3 digits are "234," which is incorrect.
- After typing "5," the most recent 3 digits are "345," which is correct, and the safe unlocks.

The objective is to return the shortest string that, when entered, will eventually unlock the safe. For example:

- Input: $n = 1$, $k = 2$
- Output: "10"
- Explanation: The password is a single digit, so entering either "01" or "10" would unlock the safe.

The problem can be summarized as generating a minimal-length string that includes every possible combination of n digits, where each digit falls within the range $[0, k-1]$. The safe checks only the most recent n digits entered at any point in the input sequence, so the goal is to unlock the safe by ensuring that the sequence contains the correct password as a substring. The assumptions made for this problem are:

1. The values of n (the length of the password) and k (the digit range) are provided by the user.
2. The sequence should be generated programmatically to ensure both efficiency and correctness.

INTRODUCTION:

The need to secure valuable assets or information has been a fundamental concern throughout human history. With the evolution of technology, security mechanisms have become more sophisticated. Physical locks have largely been replaced or supplemented by electronic safes and digital password-based security systems. These advancements have introduced complex challenges in password verification, particularly when the password is validated in a segmented manner, as in the case of the safe described in this paper.

The specific challenge we address is the development of a minimal-length sequence that ensures every possible password combination of a given length appears at some point within the sequence. This problem has significant implications not only for the design of secure safes but also for broader applications in fields like cryptography, coding theory, and network security. The solution to this problem relies on the mathematical concept of De Bruijn sequences.

A De Bruijn sequence for an alphabet of size k and length n is a cyclic sequence in which every possible substring of length n appears exactly once. These

sequences are widely used in fields where exhaustive combinations of elements are required, such as digital testing, network design, and cryptography.

To solve the safe password problem, we developed a C program that generates De Bruijn sequences based on user inputs for n and k . The generated sequence guarantees the inclusion of all possible password combinations, providing a solution for unlocking the safe. This method also showcases the practical utility of combinatorial algorithms in security applications.

DE BRUIJN SEQUENCE:

A "De Bruijn sequence" refers to a cyclic sequence where every possible combination of a given set of characters appears exactly once as a contiguous substring. The sequence has several variations based on the alphabet size and the required substring length:

- ****Binary De Bruijn Sequence****: This is a sequence consisting of 0s and 1s, where every possible sequence of length n appears exactly once.
- **** k -ary De Bruijn Sequence****: This is a more general sequence that uses an alphabet of size k , where every possible substring of length n appears exactly once.

These sequences have applications across a wide range of fields, including combinatorics, network design, bioinformatics, coding theory, and testing of digital systems. Named after Dutch mathematician Nicolaas Govert de Bruijn, who first studied them in the 1940s, De Bruijn sequences are critical tools for generating exhaustive combinations efficiently.

PROGRAM IMPLEMENTATION

The following C program generates a De Bruijn sequence for user-defined values of n and k . It uses a recursive approach to construct the sequence, ensuring that all possible n -length digit combinations are included exactly once.

```
``c
#include <stdio.h>
#include <stdlib.h>

void generateDeBruijn(int n, int k, char *a, char *result,
int t, int p, int *idx) {
    if (t > n) {
        if (n % p == 0) {
            for (int j = 1; j <= p; j++) {
                result[(*idx)++] = a[j] + '0';
            }
        }
    }
}
```

```

    }
    result[( *idx)++] = ' ';
}
} else {
    a[t] = a[t - p];
    generateDeBruijn(n, k, a, result, t + 1, p, idx);
    for (int j = a[t - p] + 1; j < k; j++) {
        a[t] = j;
        generateDeBruijn(n, k, a, result, t + 1, t, idx);
    }
}
}
}

```

```

char* deBruijnSequence(int n, int k) {
    int size = 1;
    for (int i = 0; i < n; i++) {
        size *= k;
    }
    size += n - 1 + size; // Adding space for gaps

    char *a = (char *)malloc((n + 1) * sizeof(char));
    char *result = (char *)malloc((size + 1) *
sizeof(char));
    int idx = 0;
    for (int i = 0; i <= n; i++) {
        a[i] = 0;
    }
}

```



```

    }
    generateDeBruijn(n, k, a, result, 1, 1, &idx);
    result[size] = '\0';
    free(a);
    return result;
}

int main() {
    int n, k;

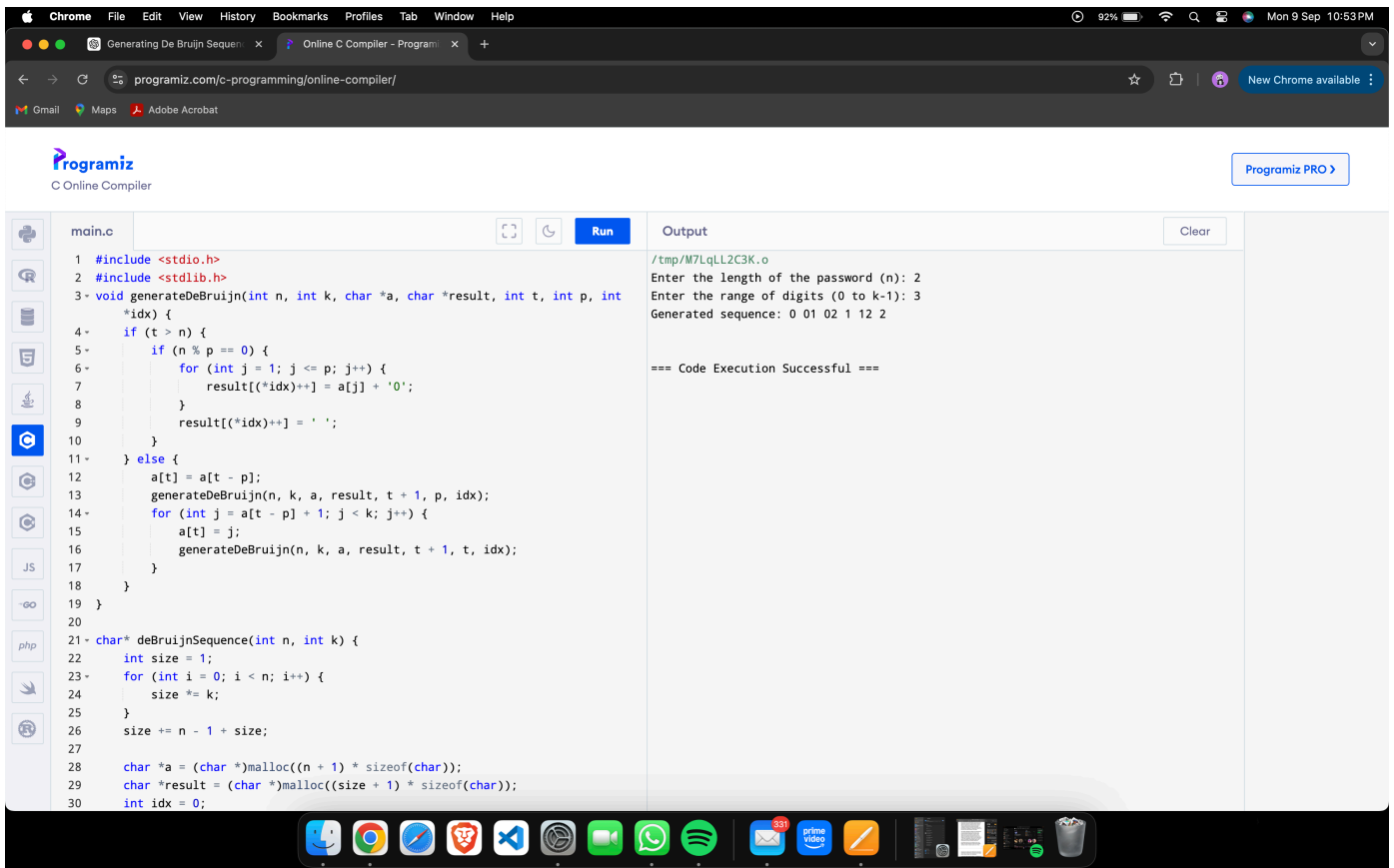
    printf("Enter the length of the password (n): ");
    scanf("%d", &n);
    printf("Enter the range of digits (0 to k-1): ");
    scanf("%d", &k);

    char *sequence = deBruijnSequence(n, k);
    printf("Generated sequence: %s\n", sequence);

    free(sequence);
    return 0;
}

```

Result:



The screenshot shows a web browser window with the URL `programiz.com/c-programming/online-compiler/`. The page title is "Programiz C Online Compiler". The code editor displays a C program for generating a De Bruijn sequence. The program includes `<stdio.h>` and `<stdlib.h>`. It defines a recursive function `generateDeBruijn` and a function `deBruijnSequence`. The `main` function prompts the user for the length of the password (`n`) and the range of digits (`k`), then calls `deBruijnSequence` to generate the sequence. The output window shows the user input: `Enter the length of the password (n): 2` and `Enter the range of digits (0 to k-1): 3`. The generated sequence is `0 01 02 1 12 2`. The output also includes the message `=== Code Execution Successful ===`.

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 void generateDeBruijn(int n, int k, char *a, char *result, int t, int p, int
   *idx) {
4     if (t > n) {
5         if (n % p == 0) {
6             for (int j = 1; j <= p; j++) {
7                 result[(*idx)++] = a[j] + '0';
8             }
9             result[(*idx)++] = ' ';
10        }
11    } else {
12        a[t] = a[t - p];
13        generateDeBruijn(n, k, a, result, t + 1, p, idx);
14        for (int j = a[t - p] + 1; j < k; j++) {
15            a[t] = j;
16            generateDeBruijn(n, k, a, result, t + 1, t, idx);
17        }
18    }
19 }
20
21 char* deBruijnSequence(int n, int k) {
22     int size = 1;
23     for (int i = 0; i < n; i++) {
24         size *= k;
25     }
26     size += n - 1 + size;
27
28     char *a = (char *)malloc((n + 1) * sizeof(char));
29     char *result = (char *)malloc((size + 1) * sizeof(char));
30     int idx = 0;
```

Output

```
/tmp/M7LqLL2C3K.o
Enter the length of the password (n): 2
Enter the range of digits (0 to k-1): 3
Generated sequence: 0 01 02 1 12 2

=== Code Execution Successful ===
```

COMPLEXITY ANALYSIS:

The time complexity of the De Bruijn sequence generation algorithm is primarily dictated by the recursive nature of the function. Each recursive call processes a possible character in the sequence, resulting in a total of k^n possible substrings for an alphabet of size k and a substring length of n . This leads to a time complexity of $O(k^n)$, ensuring that all possible substrings are generated.

The space complexity is proportional to $k^n + n - 1$ due to the storage of the entire sequence and the auxiliary arrays required for recursion. Therefore, the space complexity is $O(k^n + n)$.

BEST CASE, WORST CASE, AND AVERAGE CASE ANALYSIS:

- **Best Case:** The algorithm generates the sequence with minimal backtracking, efficiently creating the valid sequence without exploring unnecessary possibilities. The time complexity in this scenario remains $O(k^n)$, with space complexity proportional to $O(k^n + n)$.
- **Worst Case:** The worst-case complexity arises when the algorithm must explore a large portion of the search space due to repeated invalid sequences. The time complexity is still $O(k^n)$, but with higher constant factors due to increased checks and recursive calls.

- **Average Case:** The average case is closer to the best-case scenario, with some backtracking but overall efficient generation of the sequence. The average time complexity remains $O(k^n)$.

FUTURE SCOPE:

There are several directions for future research and development. Optimizations could be made to improve the efficiency of the algorithm, particularly for larger values of n and k . Parallel processing could be explored to speed up sequence generation.

Additionally, a more user-friendly interface and graphical visualizations could enhance the overall experience of generating De Bruijn sequences. Further research may focus on the application of De Bruijn sequences in cryptography, coding theory, and network design.

CONCLUSION:

The problem of generating a minimal-length sequence to ensure all possible password combinations are included can be effectively addressed using De Bruijn sequences.