De Bruijn Sequence

Your task is to construct a minimum-length bit string that contains all possible substrings of length n.

For example, when n=2, the string 00110 is a valid solution, because its substrings of length 2 are 00, 01, 10 and 11.

Input

The only input line has an integer n.

Output

Print a minimum-length bit string that contains all substrings of length n.
You can print any valid solution.

Constraints

1 <= n <= 15

Example

Input:

2

Output:

00110

What is a De Bruijn sequence?

-> Pick an alphabet of size k. For example, if k=2 the alphabet is {0,1}.

-> Pick a length n. We want a string that contains every possible length-n string (over that alphabet) exactly once as a contiguous substring.

Example: n = 3, k = 2

so we have characters as {0,1} as k = 2

possible substrings we can make from it of length 3 are

000, 001, 010, 011, 100, 101, 110, 111

Now a De Bruijn Sequence is one in which we combine all of them as the sequence for it would be

0001011100

0 0 0 1 0 1 1 1 0 0

There are k^n different length-n strings, so the De Bruijn string's length is k^n + n − 1.

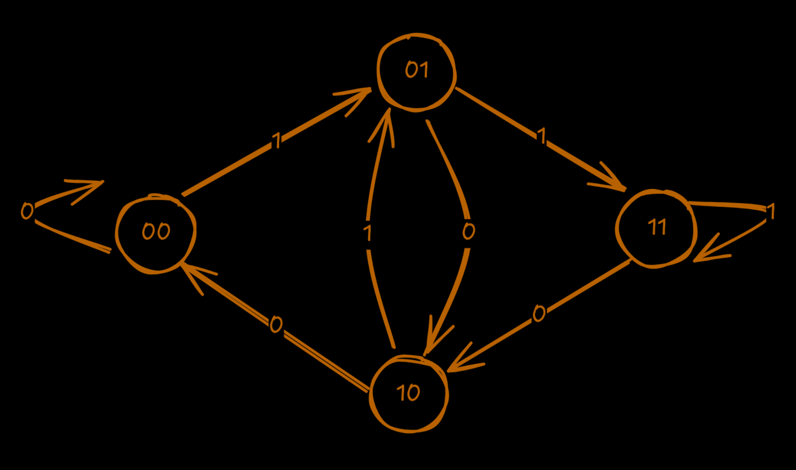
Why graphs and Eulerian paths help

The idea is to construct a graph where each node contains a string of n − 1 characters and each edge adds one character to the string.

The label on that edge is the last character you appended

for n = 3 what are all the substrings of length n-1
these are 00,01,10,11

so make all of these as nodes and then each edge will tell us the character we should append to go to next substring



Now: if you can find a path that uses every edge exactly once (an Eulerian trail/circuit), then:

Write down the starting node (it gives the first n−1 characters), then write the character on each edge as you traverse edges in order.

That produces a string that contains every length-n substring exactly once — i.e., a De Bruijn sequence.

Why does the Eulerian path exist? Because in this graph each node has exactly k outgoing edges and exactly k incoming edges (perfect balance), so an Eulerian circuit exists.

How to build the graph in code

Represent each (n-1)-string as an integer node 0..(k^(n-1)−1). Think of it as the number in base-k.

From node u and digit d (0..k-1), the next node v is:

v = (u * k + d) % k^(n-1)

This simulates shifting left by one base-k digit and appending d, then keeping only the last n-1 digits (the remainder does that).

Each such move corresponds to edge labeled d. There are k outgoing edges per node.

We can store this in a vector of vectors, where graph[u] stores pairs (nextNode, digit).

Formula to compute nextNode without strings:

nextNode = (u * k + d) % (k^(n-1))

Why?

u * k + d is like shifting the digits of u left in base-k and adding new digit d at the end.

% (k^(n-1)) keeps only the last n-1 digits.

Example in numbers:

Let's say we have n=3, k=2, and we're at node u=1 (binary 01):

Try d=0: (1 * 2 + 0) = 2 → binary "10" → node 2

Try d=1: (1 * 2 + 1) = 3 → binary "11" → node 3

How to find the Eulerian circuit (Hierholzer's algorithm)

Start at node 0 (represents string of n-1 copies of the first alphabet character).

Keep walking along unused outgoing edges until you cannot continue.

When stuck, backtrack and add nodes/edge labels to the circuit in the reverse order you finish them (this is exactly Hierholzer).

After finishing, reverse the recorded edge labels: that's the order of digits to append.

Finally: output = starting node string (n−1 chars) + the sequence of digits recorded (k^n digits). Total length = k^n + n − 1.

Algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

// Generate a De Bruijn sequence for alphabet size k and substring length n.
// alphabet supports up to 36 symbols: '0'..'9' then 'A'..'Z'.
string de_bruijn(int k, int n) {
    const string alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    if (k <= 0 || n <= 0) return "";
    if (k > (int)alphabet.size()) {
        cerr << "k too large. Max supported is " << alphabet.size() << "\n";
        return "";
    }

    // Special case n == 1: sequence is simply all alphabet symbols once.
    if (n == 1) {
        string s;
        for (int d = 0; d < k; ++d) s.push_back(alphabet[d]);
        return s;
    }

    // number of nodes = k^(n-1)
    long long numNodes = 1;
    for (int i = 0; i < n - 1; ++i) numNodes *= k;

    // 'nextEdge[v]' is how many outgoing edges from v we've already used (range 0..k)
    vector<int> nextEdge(numNodes, 0);

    // we'll simulate a stack of pairs (node, digitUsedToEnterThisNode)
    // for the starting node we use digit = -1 (no incoming digit)
    vector<pair<int,int>> stack;
    stack.reserve(numNodes * k + 5);
    stack.push_back({0, -1});

    // circuitDigits collects the digits (edge labels) in reverse order as we backtrack
    vector<int> circuitDigits;
    circuitDigits.reserve((size_t)pow(k, n)); // optional reserve

    while (!stack.empty()) {
        int v = stack.back().first;
        if (nextEdge[v] < k) {
            // take one unused outgoing edge labeled 'd'
            int d = nextEdge[v]++;
            long long u = ( (long long)v * k + d ) % numNodes; // next node
            stack.push_back({(int)u, d});
        } else {
            // no more outgoing edges unused from v, backtrack
            auto p = stack.back();
            stack.pop_back();
            if (p.second != -1) {
                // record the digit that was used to enter this node
                circuitDigits.push_back(p.second);
            }
        }
    }

    // circuitDigits are in reverse, so reverse them
    reverse(circuitDigits.begin(), circuitDigits.end());

    // build final string: starting node (n-1 copies of the first alphabet symbol)
    // plus all k^n digits we recorded.
    string result;
    result.reserve(circuitDigits.size() + (n - 1));
    for (int i = 0; i < n - 1; ++i) result.push_back(alphabet[0]);
    for (int d : circuitDigits) result.push_back(alphabet[d]);

    return result;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int k, n;
    cin >> n;
    k = 2;

    string s = de_bruijn(k, n);
    cout << s << "\n";
    return 0;
}
```

Hierholzer's algorithm

Build final string

Time Complexity

O(k^n) — you must visit each of the k^n edges once.

Space Complexity

O(k^(n−1)) for the nodes' next-edge counters + O(k^n) for output.