Competitive Programming

Backtracking

B.Bhuvaneswaran, AP (SG) / CSE

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI
ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

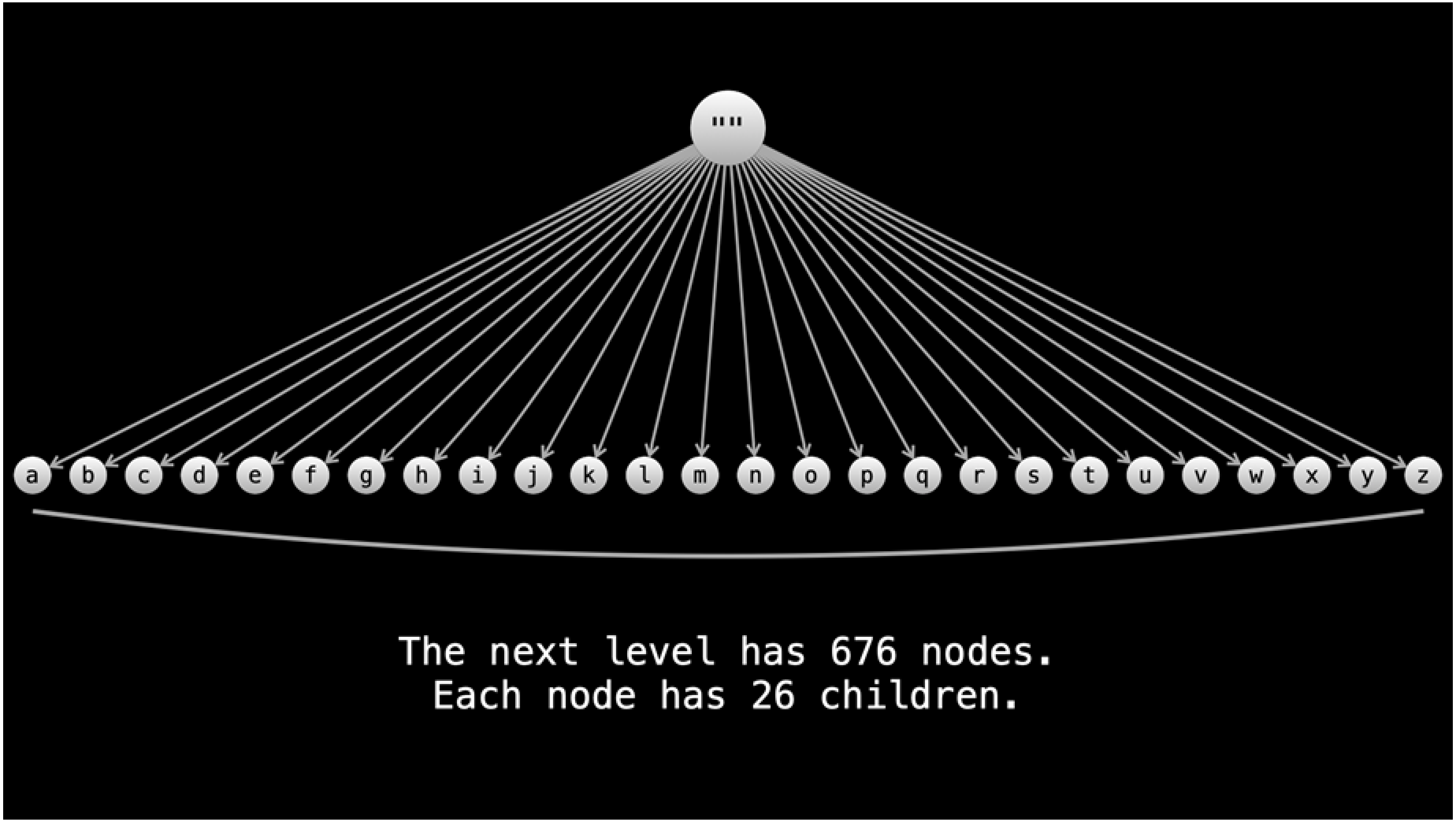
Backtracking

- The most brute force way to solve a problem is through exhaustive search.
- Generate all possibilities and then check each of them for a solution.

Example

- Imagine you had the letters a-z and were asked to generate strings of length n using the letters.
- There are 26^n possibilities, as each of the n letters could be a-z.
- You can imagine all possibilities as a tree.
- The root is the empty string "", and then all nodes have 26 children, with the path from the root representing the string being built.
- So if you started at the root and went to the "g" node, then from that node went to the "p" node, that would represent the string "gp".
- The depth of the tree is n, and the leaf nodes represent answers.

Example


""
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
The next level has 676 nodes.
Each node has 26 children.

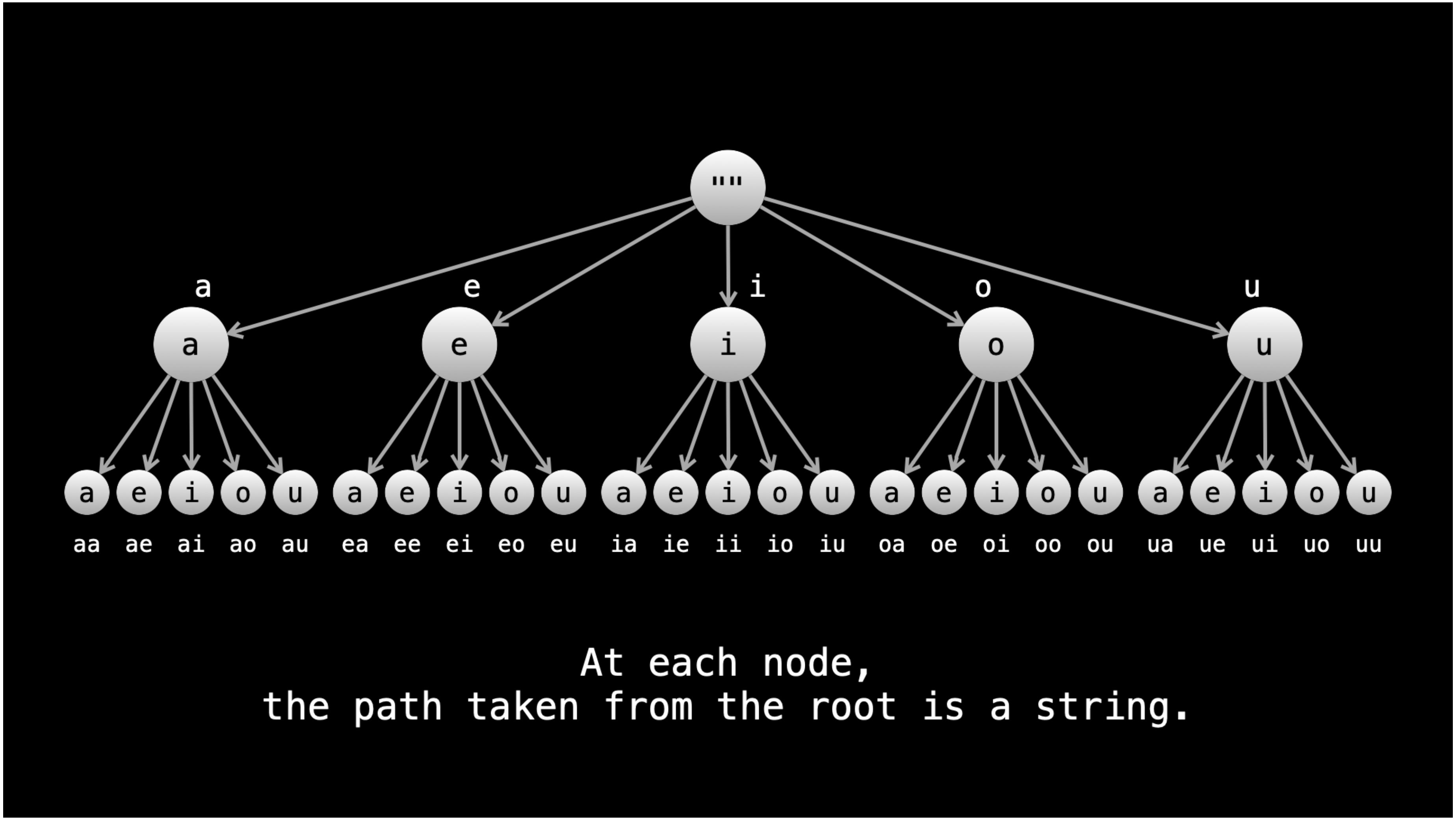
Example

- Now, let's say we added a constraint so that instead of all solutions of length n, we only want ones that meet the constraint.
- An exhaustive search would still generate all strings of length n as "candidates", and then check each one for the constraint.
- This would have a time complexity of $O(k \cdot 26^n)$, where k is the work it costs to check if a string meets the constraint.
- This time complexity is ridiculously slow.

Backtracking

- Backtracking is an optimization that involves abandoning a "path" once it is determined that the path cannot lead to a solution.
- The idea is similar to binary search trees - if you're looking for a value x, and the root node has a value greater than x, then you know you can ignore the entire right subtree.
- Because the number of nodes in each subtree is exponential relative to the depth, backtracking can save huge amounts of computation.
- Imagine if the constraint was that the string could only have vowels - an exhaustive search would still generate all 26^n strings, and then check each one for if it only had vowels.
- With backtracking, we discard all subtrees that have non-vowels, improving from $O(26^n)$ candidates to $O(5^n)$.

Backtracking



Note

- Abandoning a path is also sometimes called "pruning".
- To summarize the difference between exhaustive search and backtracking:
  - In an exhaustive search, we generate all possibilities and then check them for solutions.
  - In backtracking, we prune paths that cannot lead to a solution, generating far fewer possibilities.

Backtracking

- Backtracking is a great tool whenever a problem wants you to find all of something, or there isn't a clear way to find a solution without checking all logical possibilities.
- A strong hint that you should use backtracking is if the input constraints are very small (n <= ~15), as backtracking algorithms usually have exponential time complexities.

Implementation

- Backtracking is almost always implemented with recursion - it really doesn't make sense to do it iteratively.
- In most backtracking problems, you will be building something, either directly (like modifying an array) or indirectly (using variables to represent some state).

Pseudocode

```
// let curr represent the thing you are building
// it could be an array or a combination of variables
function backtrack(curr)
{
        if (base case)
        {
                Increment or add to answer
                return
        }
        for (iterate over input)
        {
                Modify curr
                backtrack(curr)
                Undo whatever modification was done to curr
        }
}
```

Explanation

- Let's think back to the analogy of possibilities being represented by a tree.
- Each call to the function backtrack represents a node in the tree. Each iteration in the for loop represents a child of the current node, and calling backtrack in that loop represents moving to a child.
- The line where you undo the modifications is the "backtracking" step and is equivalent to moving back up the tree from a child to its parent.

Explanation

- At any given node, the path from the root to the node represents a candidate that is being built.
- The leaf nodes are complete solutions and represent when the base case is reached.
- The root of this tree is an empty candidate and represents the scope that the original backtrack call is being made from.

Rat in a Maze

- Let's Consider a rat placed at (0, 0) in an N * N square matrix.
- It must arrive at the destination on time (N - 1, N - 1).
- Find all possible routes for the rat to take from source to destination.
- The rat can move in the following directions: 'U' (up), 'D' (down), 'L' (left), and 'R' (right) (right).
- A cell in the matrix with a value of 0 indicates that it is blocked and the rat cannot move to it, whereas a cell with a value of 1 indicates that the rat can travel through it.

Example 1

- Input:
  - 4
  - 1 0 0 0
  - 1 1 1 1
  - 0 1 0 0
- Output
  - 1 1 1 1
  - 1 0 0 0
  - 1 1 0 0
  - 0 1 0 0
  - 0 1 1 1

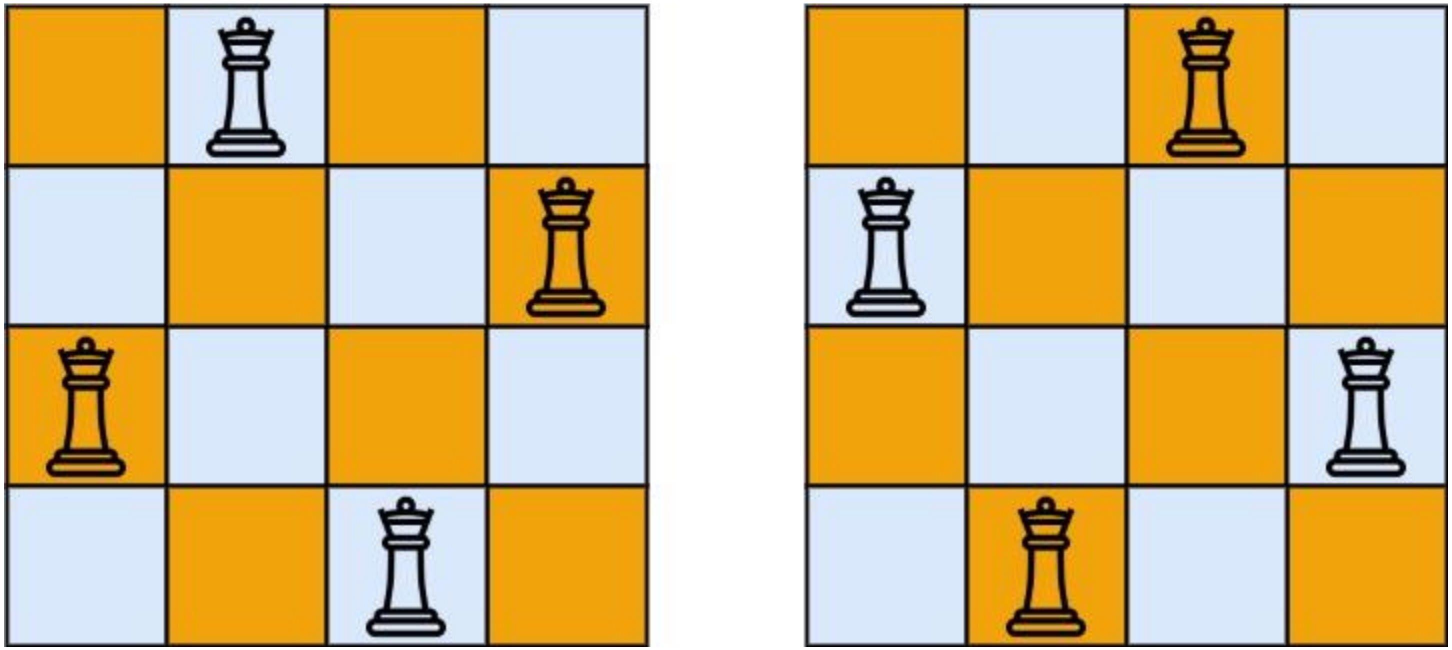
Solution

```
int solvemaze(int r, int c)
{
        if (r == size - 1 && c == size - 1)
        {
                solution[r][c] = 1;
                return 1;
        }
        if (r >= 0 && c >= 0 && r < size && c < size && solution[r][c] == 0 && maze[r][c] == 1)
        {
                solution[r][c] = 1;
                if (solvemaze(r, c + 1))
                        return 1;
                if (solvemaze(r + 1, c))
                        return 1;
                solution[r][c] = 0;
                return 0;
        }
        return 0;
}
```

N-Queens

- The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.
- Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.
- Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1



Example 1

- Input:
  - n = 4
- Output:
  - [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
- Explanation:
  - There exist two distinct solutions to the 4-queens puzzle as shown above

Example 2

- Input:
  - n = 1
- Output:
  - [["Q"]]

Constraints

- $1 <= n <= 9$

Solution

```
void Queen(int row, int n)
{
        int col;
        for (col = 1; col <= n; col++)
        {
                if (place(row, col))
                {
                        board[row] = col;
                        if (row == n)
                                print(n);
                        else
                                Queen(row + 1, n);
                }
        }
}
```

Solution

```
int place(int row, int col)
{
        for (int i = 1; i <= row - 1; i++)
        {
                if (board[i] == col)
                        return 0;
                if (abs(board[i] - col) == abs(i - row))
                        return 0;
        }
        return 1;
}
```

Solution

```
void print(int n)
{
        for (int i = n; i >= 1; i--)
        {
                for (int j = 1; j <= n; j++)
                {
                        if (board[i] == j)
                                printf("Q");
                        else
                                printf(".");
                }
                printf("\n");
        }
}
```

Queries?

Thank You…!