BackTracking

Gyanu Mayank

17th February, 2023

1. Problems Based on BackTracking

Problems

1. **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 queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.
   Input: n = 4
   Output: [[".Q..","...Q", "Q...","..Q."],["..Q.", "Q...","...Q",".Q.."]]
   Approach

chessboard and find the right arrangement where all the n queens can be placed on the n*n grid.

Implementation

```
#include <bits/stdc++.h>

using namespace std;
class Solution {
  public:
    bool isSafe1(int row, int col, vector <

      int duprow = row;
      int dupcol = col;

      while (row >= 0 && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        row--;
        col--;
      }

      col = dupcol;
      row = duprow;
      while (col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        col--;
      }

      row = duprow;
      col = dupcol;
      while (row < n && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
      }
      return true;
    }

  public:
    void solve(int col, vector < string > &
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (isSafe1(row, col, board, n)) {
          board[row][col] = 'Q';
          solve(col + 1, board, ans, n);
          board[row][col] = '.';
        }
      }
    }

  public:
    vector < vector < string >> solveNQueer
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      solve(0, board, ans, n);
      return ans;
    }
};
int main() {
  int n;
  cin>>n;
  Solution obj;
```

```
    cout << "Arrangement " << i + 1 << "\n
    for (int j = 0; j < ans[0].size(); j++)
      cout << ans[i][j];
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Optimised Approach

In the previous issafe function, we need o(N) for a row, o(N) for the column, and o(N) for the diagonal. Here, we will use hashing to maintain a list to check whether that position can be the right one or not.

Implementation

```
#include <bits/stdc++.h>

using namespace std;
class Solution {
  public:
    void solve(int col, vector < string > &
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (leftrow[row] == 0 && lowerDiago
          board[row][col] = 'Q';
          leftrow[row] = 1;
          lowerDiagonal[row + col] = 1;
          upperDiagonal[n - 1 + col - row]
          leftrow[row] = 0;
          lowerDiagonal[row + col] = 0;
          upperDiagonal[n - 1 + col - row]
        }
      }
    }

  public:
    vector < vector < string >> solveNQueer
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      vector < int > leftrow(n, 0), upperDi
      solve(0, board, ans, leftrow, upperDi
      return ans;
    }
};
int main() {
  int n;
  cin>>n;
  Solution obj;
  vector < vector < string >> ans = obj.sol
  for (int i = 0; i < ans.size(); i++) {
    cout << "Arrangement " << i + 1 << "\n'
    for (int j = 0; j < ans[0].size(); j++)
      cout << ans[i][j];
      cout << endl;
    }
    cout << endl;
  }
```

Time Complexity: O(N! * N)
Space Complexity: O(N)

2. [Rat in a Maze]

Consider a rat placed at (0, 0) in a square matrix of order N * N. It has to reach the destination at (N - 1, N - 1). Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are 'U'(up), 'D'(down), 'L' (left), and 'R' (right). Value 0 at a cell in the matrix represents that it is blocked and the rat cannot move to it while value 1 at a cell in the matrix represents that the rat can travel through it.
Example
Input: N = 4 m[][] = {{1, 0, 0, 0}, {1, 1, 0, 1}, {1, 1, 0, 0}, {0, 1, 1, 1}}
Output: DDRDRR DRDDRR

Approach

Start at the source(0,0) with an empty string and try every possible path i.e upwards(U), downwards(D), leftwards(L) and rightwards(R). As the answer should be in lexicographical order so it's better to try the directions in lexicographical order i.e (D, L, R, U)Declare a 2D-array named visited because the question states that a single cell should be included only once in the path, so it's important to keep track of the visited cells in a particular path. If a cell is in the path, mark it in the visited array. Also, keep a check of the "out of bound" conditions while going in a particular direction as shown in the recursion tree. While getting back, keep on unmarking the visited array for the respective direction. Also check whether there is a different path possible while getting back and if yes, then mark that cell in the visited array.

Implementation

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
  void findPathHelper(int i, int j, vector
    vector < vector < int >> & vis) {
    if (i == n - 1 && j == n - 1) {
      ans.push_back(move);
      return;
    }


    if (i + 1 < n && !vis[i + 1][j] && a[i
      vis[i][j] = 1;
      findPathHelper(i + 1, j, a, n, ans, m
      vis[i][j] = 0;
    }


    if (j - 1 >= 0 && !vis[i][j - 1] && a[i
      vis[i][j] = 1;
      findPathHelper(i, j - 1, a, n, ans, m
      vis[i][j] = 0;
    }
```

```
      vis[i][j] = 1;
      findPathHelper(i, j + 1, a, n, ans, m
      vis[i][j] = 0;
    }


    if (i - 1 >= 0 && !vis[i - 1][j] && a[i
      vis[i][j] = 1;
      findPathHelper(i - 1, j, a, n, ans, m
      vis[i][j] = 0;
    }

  }
  public:
    vector < string > findPath(vector < vec
      vector < string > ans;
      vector < vector < int >> vis(n, vecto

      if (m[0][0] == 1) findPathHelper(0, 0
      return ans;
    }
};

int main() {
  int n;
  cin>>n;
  vector<vector<int>>m(n,vector<int>(n));
  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
      cin>>m[i][j];
    }
  }

  Solution obj;
  vector < string > result = obj.findPath(m
  else
    for (int i = 0; i < result.size(); i++)
  cout << endl;

  return 0;
}
```

Time Complexity: O(4^(m*n))
Space Complexity: O(m*n)

3. Print all possible combinations of r elements in a given array of size n

Given an array of size n, generate and print all possible combinations of r elements in the array.
Example
Input: n=4 arr[]={1,2,3,4}
Output: {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} {3, 4}.

Approach

We create a temporary array 'data[]' which stores all outputs one by one. The idea is to start from the first index (index = 0) in data[], one by one fix elements at this index and recur for the remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at index 0 in data[], then recur for remaining indexes, and then we fix 2 at index 0 and recur. Finally, we fix 3 and recur for the remaining indexes. When the number of elements in data[] becomes equal to r (size of a combination), we print data[].

Implementation

```
void combinationUtil(int arr[], int data[],
                     int start, int end,
                     int index, int r);
void printCombination(int arr[], int n, int
{

  int data[r];

  combinationUtil(arr, data, 0, n - 1, 0, r
}

void combinationUtil(int arr[], int data[],
                     int start, int end,
                     int index, int r)
{

  if (index == r)
  {
    for (int j = 0; j < r; j++)
      cout << data[j] << " ";
    cout << endl;
    return;
  }

  for (int i = start; i <= end &&
                      end - i + 1 >= r - in
       i++)
  {
    data[index] = arr[i];
    combinationUtil(arr, data, i + 1,
                    end, index + 1, r);
  }
}
```

```
  int n;
  cin >> n;
  int arr[n];
  for (int i = 0; i < n; i++)
    cin >> arr[i];
  int r;
  cin >> r;
  printCombination(arr, n, r);
}
```

Optimised Approach

The element is included in the current combination (We put the element in data[] and increment the next available index in data[]) The element is excluded in the current combination (We do not put the element and do not change the index). When the number of elements in data[] becomes equal to r (size of a combination), we print it.

Implementation

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

void combinationUtil(int arr[], int n, int
                    int index, int data[],

void printCombination(int arr[], int n, int
{

    int data[r];
    combinationUtil(arr, n, r, 0, data, 0);
}
```

```
{

    if (index == r)
    {
        for (int j = 0; j < r; j++)
            cout << data[j] << " ";
        cout << endl;
        return;
    }


    if (i >= n)
        return;

    data[index] = arr[i];
    combinationUtil(arr, n, r, index + 1,

    combinationUtil(arr, n, r, index, data,
}


int main()
{
   int n;
   cin>>n;
   int arr[n];
   for(int i=0;i<n;i++) cin>>arr[i];
    int r;
  cin>>r;
    printCombination(arr, n, r);
}
```

Time Complexity: O(n^r)

Auxiliary Space : O(r)

length n, where any two successive strings differ in exactly one bit (i.e., their Hamming distance is one).
Your task is to create a Gray code for a given length n.
Example
Input:
2
Output:
00
01
11
10

Approach

The idea is to recursively append the bits 0 and 1 each time until the number of bits is not equal to N.
Base Condition: The base case for this problem will be when the value of N = 0 or 1.
Recursive Condition: Otherwise, for any value greater than 1, recursively generate the gray codes of the N – 1 bits and then for each of the gray codes generated add the prefixes 0 and 1.

Implementation

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

vector<string> generateGray(int n)
{

    if (n <= 0)
        return {"0"};
```

```
      return { 0 , 1 };
    }


    vector<string> recAns=
          generateGray(n-1);
    vector<string> mainAns;


    for(int i=0;i<recAns.size();i++)
    {
      string s=recAns[i];
      mainAns.push_back("0"+s);
    }


    for(int i=recAns.size()-1;i>=0;i--)
    {
       string s=recAns[i];
       mainAns.push_back("1"+s);
    }
    return mainAns;
}


void generateGrayarr(int n)
{
    vector<string> arr;
    arr=generateGray(n);

    for (int i = 0 ; i < arr.size();
         i++ )
        cout << arr[i] << endl;
}
```

```
{
    int n;
    cin>>n;
    generateGrayarr(n);
    return 0;
}
```

Time Complexity: O(2^N)
Space Complexity: O(2^N)

5. **Palindrome Partitioning**
You are given a string s, and partition it in such a way that every substring is a palindrome. Return all such palindromic partitions of s.
Example
Input: s = “aab”
Output: [["a", "a", "b"], ["aa", "b"]]
Approach
The initial idea will be to make partitions to generate substrings and check if the substring generated out of the partition will be a palindrome. Partitioning means we would end up generating every substring and checking for palindrome at every step. Since this is a repetitive task being done again and again, at this point we should think of recursion. The recursion continues until the entire string is exhausted. After partitioning, every palindromic substring is inserted in a data structure When the base case has reached the list of palindromes generated
d during that recursion call is inserted in a vector of vectors/list of lists. Say s="aabb" We consider substrings starting from the 0th

make a partition after 1st index. Beyond this point, other substrings starting from index 0 are “aab” and “aabb”. These are not palindromes, hence no more. partitions are possible. The strings remaining on the right side of the partition are used as input to make recursive calls.

Implementation

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
  public:
    vector < vector < string >> partition(s
      vector < vector < string > > res;
      vector < string > path;
      partitionHelper(0, s, path, res);
      return res;
    }

  void partitionHelper(int index, string s,
    vector < vector < string > > & res) {
    if (index == s.size()) {
      res.push_back(path);
      return;
    }
    for (int i = index; i < s.size(); ++i)
      if (isPalindrome(s, index, i)) {
        path.push_back(s.substr(index, i -
        partitionHelper(i + 1, s, path, res
        path.pop_back();
      }
  bool isPalindrome(string s, int start, ir
    while (start <= end) {
      if (s[start++] != s[end--])
        return false;
    }
    return true;
  }
};
int main() {
  string s;
  cin>>s;
  Solution obj;
  vector < vector < string >> ans = obj.par
  int n = ans.size();
  cout << "The Palindromic partitions are :
  cout << " [ ";
  for (auto i: ans) {
    cout << "[ ";
    for (auto j: i) {
      cout << j << " ";
    }
    cout << "] ";
  }
  cout << "]";

  return 0;
}
```

Time Complexity: O((2^n) *k*(n/2))
Space Complexity: O(k * x)

6. Generate Parentheses

Given n pairs of parentheses, write a function

Example
Input: n=2
Output: {}{} {{}}

Approach 1 Recursive

Do recursion and generate all the possible sequences and also keep checking if open == close and when the size of the string becomes 2*n then push that string in the answer vector.

Implementation

```
#include <bits/stdc++.h>

using namespace std;
vector<string>ans;
void generate(int open, int close, int n, s
{
  if (s.size() == 2 * n)
    ans.push_back(s);
  if (open < n)
    generate(open + 1, close, n, s + "(");
  if (close < open)
    generate(open, close + 1, n, s + ")");
}
vector<string> generateParenthesis(int n)
{
  generate(0, 0, n, "");
  return ans;
}
int main()
{
  int n;
  cin >> n;
  generateParenthesis(n);
```

Approach using Backtracking

When the length of our string has reached the maximum length(`n*2`), we stop with the recursion for that case and that is our base case. On observing carefully we find that there are two conditions present: For adding **`(`**: If the number of opening brackets(`open`) is less than the given length(`n`) i.e.
if `max<n`, then we can add **`(`**, else not.For adding **`)`**: If the number of close brackets(`close`) is less than the opening brackets(`open`), i.e.
if `open<close`, we can add **`)`**, else not.

Implementation

```
#include <bits/stdc++.h>

using namespace std;
vector<string> valid;
void generate(string &s,int o,int c){
if(o==0&&c==0){
    valid.push_back(s);
    return;
}
if(o>0){
    s.push_back('(');
    generate(s,o-1,c);
    s.pop_back();
}
if(c>0){
    if(o<c){
        s.push_back(')');
    }
}
}
    vector<string> generateParenthesis(int
        string s;
        generate(s,n,n);
        return valid;
    }

int main()
{
  int n;
  cin >> n;
  generateParenthesis(n);
  return 0;
}
```

Time complexity: O(2^n)
Space complexity: O(2^n + n)

BOSSCODER ACADEMY UPSKILL WITH US

Bosscoder had everything I was looking for

Read

Date: 12th May, 2023

"Bosscoder helped me clear my basics of DSA and System Design"
Rachit Arora
Head of Engineering
SOPTLE

Bosscoder helped me clear my basics of DSA and System Design

Read

Date: 5th May, 2023

View All blogs

Helping ambitious learners upskill themselves & shift to top product based companies.

Lets hear all about it.

Who are we

About us
Blog
Attend a FREE Event
Privacy Policy
Terms & Condition
Pricing and Refund Policy

Contact Us

Email: ask@bosscoderacademy.com

Follow us on

LinkedIn
Youtube
Instagram
Telegram
Reviews on Quora