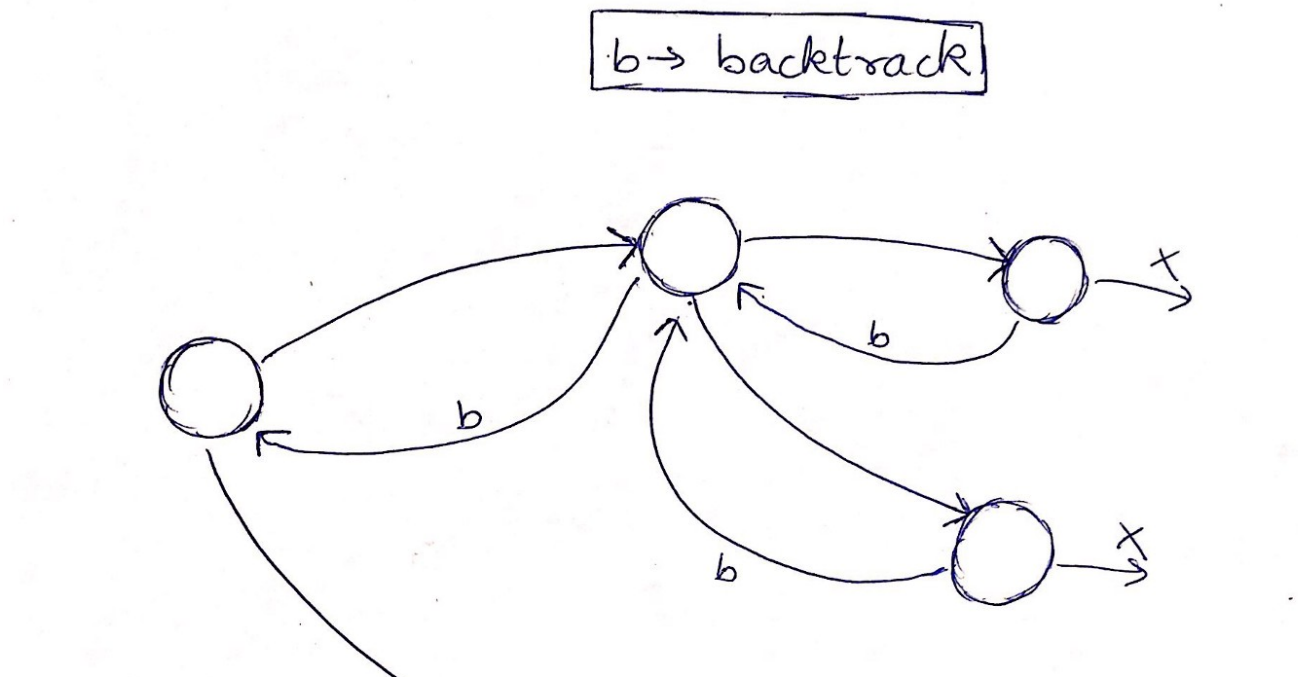
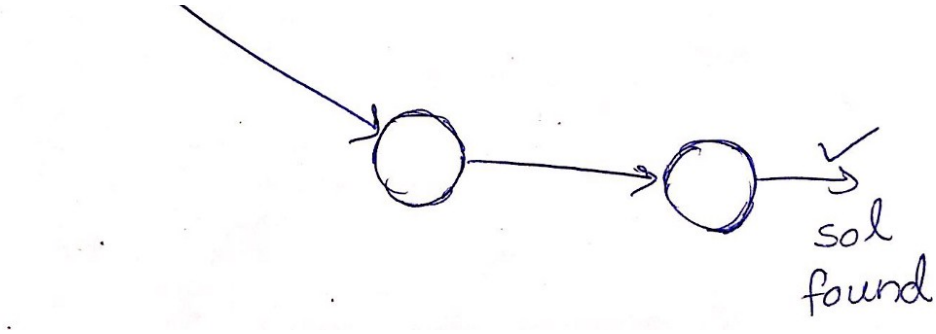
csgator Follow
Feb 21, 2018 · 5 min read

Leetcode Pattern 3 | Backtracking

A very important tool to have in our arsenal is backtracking, it is all about knowing when to stop and step back to explore other possible solutions. In today's post we'll explore the common pattern in solving backtracking problems and set up the stage to dive into dynamic programming (DP) problems next. It is amusing how a small change in the problem can change the solution from DP to backtracking and understanding this will help us save time.

Backtracking can be seen as an optimized way to brute force. Brute force approaches evaluate every possibility. In backtracking you stop evaluating a possibility as soon it breaks some constraint provided in the problem, take a step back and keep trying other possible cases, see if those lead to a valid solution.





So in fact, it's kinda like a depth-first search(DFS) with an added constraint that we stop exploring the subtree as soon as we know for sure that it won't lead to valid solution. The problems that can be solved using this tool generally satisfy the following criteria :

1. You are explicitly asked to return a collection of all answers.
2. You are concerned with what the actual solutions are rather than say the most optimum value of some parameter. (if it were the latter it's most likely DP or greedy).

78. Subsets

We'll use this problem to get familiar with the recursive backtracking pattern.

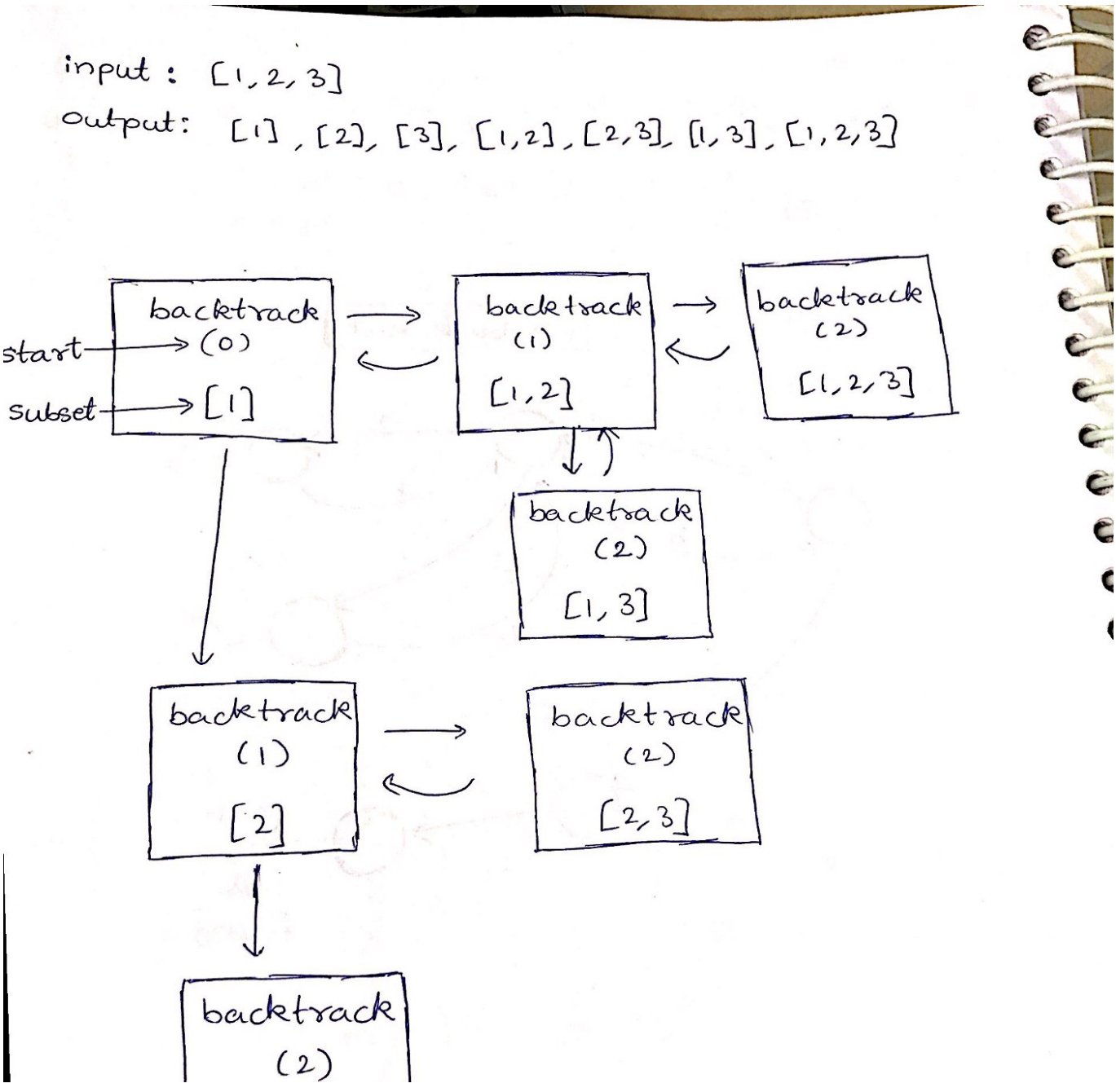
The problem is to find the powerset of a given set, so we simply need to collect all possible subsets of a set. This could be done in a variety of ways and backtracking is one way to go, also since there are no constraints provided we simply explore all cases (all subsets).

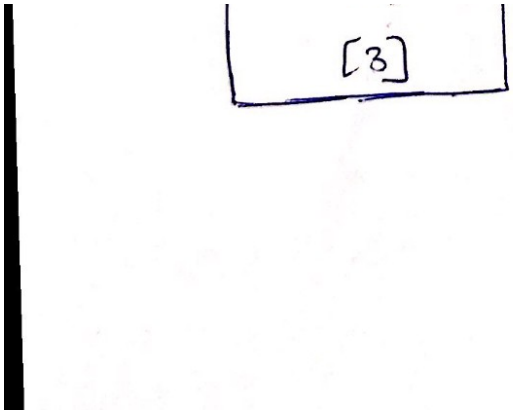
A subset can either have an element or leave it out giving rise to 2^n subsets.

```
class Solution {
private:
    vector<vector<int>> powerset; // stores all subsets
    vector<int> subset; // temporary subset which will be updated as the recursive function
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        backtrack(nums, 0);
        return powerset;
    }

    void backtrack(vector<int>& nums, int start){
        powerset.push_back(subset);
```

```
        for(int i = start; i < nums.size(); i++){
            // recording all subsets that include nums[i]
            subset.push_back(nums[i]);
            backtrack(nums, i+1);

            // remove nums[i] from the present subset and move further to explore subsets th
            subset.pop_back();
        }
    }
};
```

78-Subsets-Backtracking.cpp hosted with ❤️ by GitHub view raw





Take a moment to absorb the magic happening in the loop and that's everything you'll ever need to solve a backtracking problem. It was confusing to me at first but it's an amazing pattern. Drawing the flow of the recursive function helped me wrap my head around what is going on.

Also here is my alternative solution to the same problem which uses the partially formed output to generate the full output incrementally. It's basically deriving the complete solution from solutions of smaller problems, does it ring a bell?

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> powerset;
        powerset.push_back(vector<int>(0));

        // after each iteration of this loop we are left with the powerset of the subset
        for(int i = 0; i < nums.size(); i++){

            // append nums[i] to already recorded subsets to form new ones.
            for(int j = 0; j < (int)pow(2, i); j++){
                vector<int> subset = powerset[j];
                subset.push_back(nums[i]);

                powerset.push_back(set);
            }
        }

        return powerset;
    }
};
```

78-Subsets-Iterative.cpp hosted with ❤️ by GitHub view raw

90. Subsets II

Same problem with the added constraint that the set may contain duplicates but the output power set should not contain duplicate subsets.

Approach: Sort the given array beforehand and skip over duplicates while backtracking, essentially a simple 2 line change in the previous solution.

```
class Solution {
private:
    vector<vector<int>> powerset;
    vector<int> subset;
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end()); //change 1

        backtrack(nums, 0);
        return powerset;
    }

    void backtrack(vector<int>& nums, int start){
        powerset.push_back(subset);

        for(int i = start; i < nums.size(); i++){
            if(i > start && nums[i] == nums[i-1]) continue; //change 2, skip over duplic

            subset.push_back(nums[i]);
            backtrack(nums, i+1);

            subset.pop_back();
        }
    }
};
```

Subsets2.cpp hosted with ❤️ by GitHub view raw

39. Combination Sum

Notice that we'll have to explore many cases and there is no "smart" way to avoid that, the only smart thing we could do is to stop exploring a case as soon as we know it won't lead to the solution and so this is a backtracking problem.

The solution is entirely same as subsets solution, only with a slight modification that we have a constraint included: the sum of the final collected combination should equal target.

```cpp
class Solution {
private:
    vector<int> combination;
    vector<vector<int>> combinations;

public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        explore(candidates, 0, target);
        return combinations;
    }

    void explore(vector<int>& candidates, int start, int target){
        if(target == 0){
            combinations.push_back(combination); // yes! a valid solution found
            return;
        }
        if(target < 0) return; // this is when we lose hope and backtrack :(

        for(int i = start; i < candidates.size(); i++){
            // explore all solutions using candidates[i] at least once
            combination.push_back(candidates[i]);
            explore(candidates, i, target - candidates[i]);

            // explore solutions that don't use candidates[i]
            combination.pop_back();
        }
    }
};
```

39-Combination-Sum.cpp hosted with ❤️ by GitHub view raw

40. Combination Sum II

This problem bears the same relation to the previous problem as subsets-2 had with subsets, as such it should be no surprise that the same strategy works!

```cpp
class Solution {
private:
    vector<int> combination;
```

```cpp
    vector<int> combination;
    vector<vector<int>> combinations;
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());

        explore(candidates, 0, target);
        return combinations;
    }

    void explore(vector<int>& candidates, int start, int target){
        if(target == 0){
            combinations.push_back(combination);
            return;
        }
        if(target < 0) return;

        for(int i = start; i < candidates.size(); i++){
            if(i > start && candidates[i] == candidates[i-1]) continue; // skip duplicat

            // explore all solutions using candidates[i] exactly once
            combination.push_back(candidates[i]);
            explore(candidates, i+1, target - candidates[i]);

            // explore solutions that don't use candidates[i]
            combination.pop_back();
        }
    }
};
```

Combinations-2.cpp hosted with ❤️ by GitHub view raw

Honestly, visualizing the flow of the recursive function above is kinda trippy. Once you get comfortable writing this back and forth flow, backtracking problems are really easy. Also as I was writing this article I had an idea to make it simpler, so here is a simpler version of the subsets solution.(could be extended for other solutions in this post as well)

```cpp
class Solution {
private:
    vector<vector<int>> powerset;
    vector<int> subset;
public:
```

```
    vector<vector<int>> subsets(vector<int>& nums) {
        backtrack(nums, 0);
        return powerset;
    }

    void backtrack(vector<int>& nums, int start){
        if(start == nums.size()) powerset.push_back(subset);
        else{
            subset.push_back(nums[start]);
            backtrack(nums, start+1);

            subset.pop_back();
            backtrack(nums, start+1);
        }
    }
};
```

78-Subsets-Backtracking-Simplified.cpp hosted with ❤️ by GitHub view raw

Time for one final problem without which our discussion on backtracking would be considered incomplete, one of the classic computer science problems which gave birth to this paradigm.

51. N-Queens

We try placing queens column by column. Place a queen, go to next column and try placing another queen such that it doesn't face a queen in the same row or diagonals (which is checked in validateSpot method), and keep going. If we encounter an invalid spot we backtrack and keep trying other spots in that column vertically.

```
class Solution {
public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> solutions;
        vector<string> board(n, string(n, '.'));

        solveBoard(solutions, board, 0, n);
        return solutions;
    }

    void solveBoard(auto& solutions, auto& board, int col, int n){
        if(col == n){
            solutions.push_back(board);
```

```
            return;
        }

        for(int row = 0; row < n; row++){
            if(validateSpot(board, row, col)){

                board[row][col] = 'Q';
                solveBoard(solutions, board, col + 1, n);
                board[row][col] = '.';

            }
        }
    }

    bool validateSpot(vector<string>& board, int row, int col){
        int n = board.size();

        // check horizontally for queens in same row
        for(int x = 1; x <= col; x++){
            if(board[row][col-x] == 'Q') return false;
        }

        // check diagonals
        for(int x = 1; row-x >= 0 && col-x >= 0; x++){
            if(board[row-x][col-x] == 'Q') return false;
        }

        for(int x = 1; row+x < n && col-x >= 0; x++){
            if(board[row+x][col-x] == 'Q') return false;
        }

        return true;
    }

    void printBoard(vector<string> board){
        for(string s : board){
            cout << s << endl;
        }
    }
};
```

51.N-Queens.cpp hosted with ❤️ by GitHub view raw

The validateSpot method can be made more efficient by using arrays to store the diagonals and rows already occupied. If you are interested, do check out this solution

Finally the point I mentioned earlier, when does a backtracking problem convert to a DP one? Let's take an example:

- Return all ways to climb stairs, jumps allowed in steps 1 -> k

Technical Interview Coding Practice - CodeFights

Need to find a software engineer job? Practice your programming skills free online and solve real technical interview...

codefights.com

- Count ways to climb stairs, jumps allowed in steps 1-> k

Climb n-th stair with all jumps from 1 to n allowed (Three Different Approaches) - GeeksforGeeks

A monkey is standing below at a staircase having N steps. Considering it can take a leap of 1 to N steps at a time...

www.geeksforgeeks.org

The first would require backtracking as we actually need all the ways, while the 2nd one could be done using DP optimally and is similar to how we optimize calculating Fibonacci numbers with memoization.

In the next post we'll see solutions to these problems as well as explore other such cases (the standard rod cutting problem vs the subsets problem above). Understanding when to use DP is in itself a major issue. At this point I would like to point out the strong bond between recursion, backtracking, depth first search, and dynamic programming. (mega pattern if you will!)

The next few posts will be on solely focused on decoding DP patterns as many of you have requested the same. Stay tuned for upcoming posts!

References:

- discuss thread on backtracking
- backtracking vs dfs
- geeksforgeeks backtracking

Programming Leetcode Algorithms Data Structures Interview