CodeMonk Dynamic Programming II

82 Dynamic Programming

In case, you are planning on reading this article, I'm going to assume that you know the basics of dynamic programming. If that isn't the case, then I strongly urge you to read **Dynamic Programming - I**.

Getting down to business. Dynamic Programming, by itself, is a pretty powerful paradigm of solving problems. But, applying the basics of of DP sometimes aren't enough. Let's review the basics of DP anyway!

There's two ways of writing a DP solution:

Bottom-Up Iterative approach

Say we want to obtain the n-th fibonacci number.
The series goes like this 0, 1, 1, 2, 3, 5, 8, 13 and so on. We know that the 0th entry is 0 and the 1st entry is 1.

We could do the following.

```
int fib[maxn];
fib[ 0 ] = 0;
fib[ 1 ] = 1;
for (int i = 2; i <= n; i++) {
    fib[i] = fib[i - 1] + fib[i - 2];
}
```

We see here that we have defined the base conditions and iteratively built up the values up to the one that we wish to obtain.

Top-Down Recursive approach with memoization

Why would we want to go with this approach? Because recursive definitions are so much more intuitive and elegant than iterative solutions, in most cases at least! Let's try to solve the same problem as we did before with a recursive approach instead.

```
int fib[maxn];
memset(fib, -1, sizeof(fib));
int solve(int n) {
    if (n == 0 || n == 1) {    // base conditions
        return fib[n] = n;
    }
    if (fib[n] != -1) {    // if memoized then return the value
        return fib[n];
    }
    return fib[n] = solve(n - 1) + solve(n - 2);    // recursive
definition
}
```

We notice here that we are simply following the mathematical definition of Fibonacci series in the last line of the function, which is pretty intuitive, and the rest of the function is simply base conditions and checking whether the result at a particular point has been memoized.

Note that any problem can be solved using either approach. We usually pick one based on how simple it is to code. If the limits for a certain problem ensure that there will not be any overflow in the recursion stack (for function calls), then we can go ahead with that approach.

There are some interesting problems that can be solved with DP, like:
edit distance between two strings
constrained travelling salesman problem (TSP)
DP on trees

Let’s look at some of these problems and develop approaches to see how DP helps us in solving problems.

Edit Distance
Edit distance is a way of quantifying how dissimilar two strings are, i.e., how many operations (add, delete or replace character) it would take to transform one string to the other. This is one of the most common variants of edit distance, also called Levenshtein distance, named after Soviet computer scientist, Vladimir Levenshtein. More formally, we have two strings, s1 and s2. There are 3 operations which can be applied to either string, namely: **insertion, deletion and replacement.**

```
int editDistance(string s1, string s2) {
    int m = s1.size();
    int n = s2.size();
    // for all i, j, dp[i][j] will hold the edit distance between
the first
    // i characters of source string and first j characters of
target string
    int dp[m + 1][n + 1];
    memset(dp, 0, sizeof(dp));
    // source can be transformed into target prefix by inserting
    // all of the characters in the prefix
for (int i = 1; i <= m; i++) {
        dp[0][i] = i;
    }
    // source prefixes can be transformed into empty string by
    // by deleting all of the characters
    for (int i = 1; i <= n; i++) {
        dp[i][0] = i;
    }
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1]; // no operation
required as characters are the same
            }
            else {
                dp[i][j] = 1 + min(dp[i - 1][j - 1],    //
substitution
                        min(dp[i][j - 1],     // insertion
                            dp[i - 1][j]));    // deletion
            }
        }
    }
    return dp[m][n];
}
```

Let’s look at the DP table when s1 = “sitting” (source string)
s2 = “kitten” (target string)

		k	**i**	**t**	**t**	**e**	**n**
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Travelling Salesman Problem

When you’re given a set of cities and the distances between the cities, you wish to travel all of the cities exactly once and cover the least distance, then this is the Travelling Salesman Problem. It is an NP-Hard problem that is important in the in combinatorial optimizations, operations research and theoretical computer science. Also, ever so often, we end up seeing variations of this problem, or problems that use the techniques used in this problem. The most important of these techniques is DP with Bitmasking.

Let's formalize this problem. There's a list of 20 cities and the distances between the **i**th and **j**th city is given in an adjacency matrix dist. You start at the **0**th city and want to finish visiting all of the cities by covering the least possible distance.

Given that you have not more than 20 cities, try and formalize a DP by incorporating bitmasking.

Please don't scroll down without thinking about it for a few minutes.

Let's consider some subset of cities, **S**. Say you have visited all of the cities in **S**. Now you want to visit some city **C** that's not in **S**. What do you want to minimize now? The sum of the distance already covered by visiting all the cities in **S** and the distance between the last city visited in **S** (let's call it **P**) and **C**.

From this we can see that a state in our DP is characterized by two variables.
The set of cities we have visited, **S** The last city visited in the set of cities, **L**
The base cases would be visiting just 1 city and that distance would be 0.
Now, we can represent the set of visited cities by using bitmasking. Each integer has 32 bits, and we have at most 20 cities. So, if the i-th bit is 1, then it has been visited, else it's unvisited.

```
const int INF = 1e9;
int shortestTSPdistance(vector<vector<int> > dist) {
    int n = dist.size();
    int lim = 1 << n;
    int dp[lim][n];
    memset(dp, INF, sizeof(dp));
    for (int i = 0; i < n; i++) {
        dp[1 << i][i] = 0;    // base case of visiting just 1 city
    }
    for (int mask = 0; mask < lim; mask++) {
        for (int last = 0; last < n; last++) {
            if (mask && (1 << last) == 0) {
                continue;
            }
            for (int curr = 0; curr < n; curr++) {
                if (mask && (1 << curr) == 0) {
                    continue;
                }
                int otherMask = mask ^ (1 << curr);
                dp[mask][curr] = min(dp[mask][curr], dp[otherMask]
[last] + dist[last][curr]);
            }
        }
    }
    int ans = INF;
    for (int i = 0; i < n; i++) {
        ans = min(ans, dp[lim - 1][i]);
    }
    return ans;
}
```

Note that in most DP with bitmasking problems, there will be a variable constrained to a value of almost ~20.

Space Complexity: **$O(n * 2^n)$**

Time Complexity: **$O(n^2 * 2^n)$**

DP on Trees

Given a tree with **N** vertices where the **i^{th}** vertex has a reward **R_i**, what is the maximum reward that can be obtained such that among the vertices we choose, no two vertices are connected by an edge?

The brute force approach involves choosing all possible valid subsets of vertices and returning the maximal reward.

We have a tree. So, let's root it at a particular node. Say the node is vertex #1. If we choose a certain vertex, then we cannot choose its children. We can still choose the grandchildren.

DP[v] = max(sum of DP[u] where u is child of v, R_v + sum of DP[u] where u is each grandchild of v)

How do we simplify this formulation? We can simplify 2 DPs, DP1 and DP2 where DP1(v) is the maximal solution for the subtree rooted at v if we include the vertex v and DP2(v) is the maximal solution for the subtree rooted at v if we don't include the vertex v.

DP1(v) = R_v + sum over DP2(u) where u is the child of v

DP2(v) = sum over max(DP1(u), DP2(u)) (we are doing max as we can include the children, but we may choose not to)

```
const int N = 1e5 + 5;
vector<int> G[N];    // graph in the form of adjacency list
int dp1[N], dp2[n];    // as we have defined
int R[N];          // reward for a given node

void dfs(int v = 1, int p = 0) {
    int sum1 = 0, sum2 = 0;
    for (auto u: G[v]) {
        if (p != u) {
            dfs(u, v);
```

```
                sum1 += dp2[u];
                sum2 += max(dp1[u], dp2[u]);
            }
        }
        dp1[v] = sum1 + R[v];
        dp2[v] = sum2;
}

int main() {
        int n;
        cin >> n;
        int u, v;
        for (int i = 0; i < n; i++) {
            cin >> u >> v;
            G[u].push_back(v);
            G[v].push_back(u);
        }
        dfs();
        cout << max(dp1[ 1 ], dp2[ 1 ]);
        return 0;
}
```

In DP with Trees, it is useful to try and formulate a DP with **DP$_i$** denoting some value related to the subtree rooted at vertex **i**.

For further reading, please see:
DP on Trees by darkshadows
DP on graphs with bitmasking

Like 0 Tweet G+

AUTHOR

Tanmay Sahay
Teaching Assistant at IIIT H...
Hyderabad
1 note

TRENDING NOTES

Python Diaries Chapter 3 Map | Filter | For-else | List Comprehension
written by Divyanshu Bansal