Last update: October 11, 2024 Original

Introduction to Dynamic Programming

The essence of dynamic programming is to avoid repeated calculation. Often, dynamic programming problems are naturally solvable by recursion. In such cases, it's easiest to write the recursive solution, then save repeated states in a lookup table. This process is known as top-down dynamic programming with memoization. That's read "memoization" (like we are writing in a memo pad) not memorization.

One of the most basic, classic examples of this process is the fibonacci sequence. It's recursive formulation is $f(n) = f(n-1) + f(n-2)$ where $n \geq 2$ and $f(0) = 0$ and $f(1) = 1$. In C++, this would be expressed as:

```
int f(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

The runtime of this recursive function is exponential - approximately $O(2^n)$ since one function call ($f(n)$) results in 2 similarly sized function calls ($f(n-1)$ and $f(n-2)$).

Speeding up Fibonacci with Dynamic Programming (Memoization)

Our recursive function currently solves fibonacci in exponential time. This means that we can only handle small input values before the problem becomes too difficult. For instance, $f(29)$ results in *over 1 million* function calls!

To increase the speed, we recognize that the number of subproblems is only $O(n)$. That is, in order to calculate $f(n)$ we only need to know $f(n-1), f(n-2), \ldots, f(0)$. Therefore, instead of recalculating these subproblems, we solve them once and then save the result in a lookup table. Subsequent calls will use this lookup table and immediately return a result, thus eliminating exponential work!

Each recursive call will check against a lookup table to see if the value has been calculated. This is done in $O(1)$ time. If we have previously calcuated it, return the result, otherwise, we calculate the function normally. The overall runtime is $O(n)$! This is an enormous improvement over our previous exponential time algorithm!

```
const int MAXN = 100;
bool found[MAXN];
int memo[MAXN];

int f(int n) {
    if (found[n]) return memo[n];
    if (n == 0) return 0;
    if (n == 1) return 1;

    found[n] = true;
    return memo[n] = f(n - 1) + f(n - 2);
}
```

With our new memoized recursive function, $f(29)$, which used to result in *over 1 million calls*, now results in *only 57* calls, nearly *20,000 times* fewer function calls! Ironically, we are now limited by our data type. $f(46)$ is the last fibonacci number that can fit into a signed 32-bit integer.

Typically, we try to save states in arrays,if possible, since the lookup time is $O(1)$ with minimal overhead. However, more generically, we can save states anyway we like. Other examples include maps (binary search trees) or unordered_maps (hash tables).

An example of this might be:

```
unordered_map<int, int> memo;
int f(int n) {
    if (memo.count(n)) return memo[n];
    if (n == 0) return 0;
    if (n == 1) return 1;

    return memo[n] = f(n - 1) + f(n - 2);
}
```

Or analogously:

```
map<int, int> memo;
int f(int n) {
    if (memo.count(n)) return memo[n];
    if (n == 0) return 0;
    if (n == 1) return 1;

    return memo[n] = f(n - 1) + f(n - 2);
}
```

Both of these will almost always be slower than the array-based version for a generic memoized recursive function. These alternative ways of saving state are primarily useful when saving vectors or strings as part of the state space.

The layman's way of analyzing the runtime of a memoized recursive function is:

$$\text{work per subproblem} * \text{number of subproblems}$$

Using a binary search tree (map in C++) to save states will technically result in $O(n \log n)$ as each lookup and insertion will take $O(\log n)$ work and with $O(n)$ unique subproblems we have $O(n \log n)$ time.

This approach is called top-down, as we can call the function with a query value and the calculation starts going from the top (queried value) down to the bottom (base cases of the recursion), and makes shortcuts via memoization on the way.

Bottom-up Dynamic Programming

Until now you've only seen top-down dynamic programming with memoization. However, we can also solve problems with bottom-up dynamic programming. Bottom up is exactly the opposite of top-down, you start at the bottom (base cases of the recursion), and extend it to more and more values.

To create a bottom-up approach for fibonacci numbers, we initilize the base cases in an array. Then, we simply use the recursive definition on array:

```
const int MAXN = 100;
int fib[MAXN];

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

    return fib[n];
}
```

Of course, as written, this is a bit silly for two reasons: Firstly, we do repeated work if we call the function more than once. Secondly, we only need to use the two previous values to calculate the current element. Therefore, we can reduce our memory from $O(n)$ to $O(1)$.

An example of a bottom up dynamic programming solution for fibonacci which uses $O(1)$ might be:

```
const int MAX_SAVE = 3;
int fib[MAX_SAVE];

int f(int n) {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i <= n; i++)
        fib[i % MAX_SAVE] = fib[(i - 1) % MAX_SAVE] + fib[(i - 2) % MAX_SAVE];
```

```
    return fib[n % MAX_SAVE];
}
```

Note that we've changed the constant from `MAXN` TO `MAX_SAVE`. This is because the total number of elements we need to have access to is only 3. It no longer scales with the size of input and is, by definition, $O(1)$ memory. Additionally, we use a common trick (using the modulo operator) only maintaining the values we need.

That's it. That's the basics of dynamic programming: Don't repeat work you've done before.

One of the tricks to getting better at dynamic programming is to study some of the classic examples.

Classic Dynamic Programming Problems

Name	Description/Example
0-1 knapsack	Given W, N, and N items with weights w_i and values v_i, what is the maximum $\sum_{i=1}^{k} v_i$ for each subset of items of size k ($1 \leq k \leq N$) while ensuring $\sum_{i=1}^{k} w_i \leq W$?
Subset Sum	Given N integers and T, determine whether there exists a subset of the given set whose elements sum up to the T.
Longest Increasing Subsequence	You are given an array containing N integers. Your task is to determine the LIS in the array, i.e., a subsequence where every element is larger than the previous one.
Counting all possible paths in a matrix.	Given N and M, count all possible distinct paths from $(1, 1)$ to (N, M), where each step is either from (i, j) to $(i + 1, j)$ or $(i, j + 1)$.
Longest Common Subsequence	You are given strings s and t. Find the length of the longest string that is a subsequence of both s and t.
Longest Path in a Directed Acyclic Graph (DAG)	Finding the longest path in Directed Acyclic Graph (DAG).
Longest Palindromic Subsequence	Finding the Longest Palindromic Subsequence (LPS) of a given string.

Name	Description/Example
Rod Cutting	Given a rod of length n units, Given an integer array cuts where cuts[i] denotes a position you should perform a cut at. The cost of one cut is the length of the rod to be cut. What is the minimum total cost of the cuts.
Edit Distance	The edit distance between two strings is the minimum number of operations required to transform one string into the other. Operations are ["Add", "Remove", "Replace"]

Related Topics

- Bitmask Dynamic Programming
- Digit Dynamic Programming
- Dynamic Programming on Trees

Of course, the most important trick is to practice.

Practice Problems

- LeetCode - 1137. N-th Tribonacci Number
- LeetCode - 118. Pascal's Triangle
- LeetCode - 1025. Divisor Game
- Codeforces - Vacations
- Codeforces - Hard problem
- Codeforces - Zuma
- LeetCode - 221. Maximal Square
- LeetCode - 1039. Minimum Score Triangulation of Polygon

Dp Contests

- Atcoder - Educational DP Contest
- CSES - Dynamic Programming

Contributors:

mhayter (80.61%) Zyad-Ayad (11.52%) jakobkogler (2.42%) ShubhamPhapale (1.82%) adamant-pwn (1.21%)
Ahmed-Elshitehi (1.21%) boredumbk (0.61%) hitarth-gg (0.61%)