Algorithms

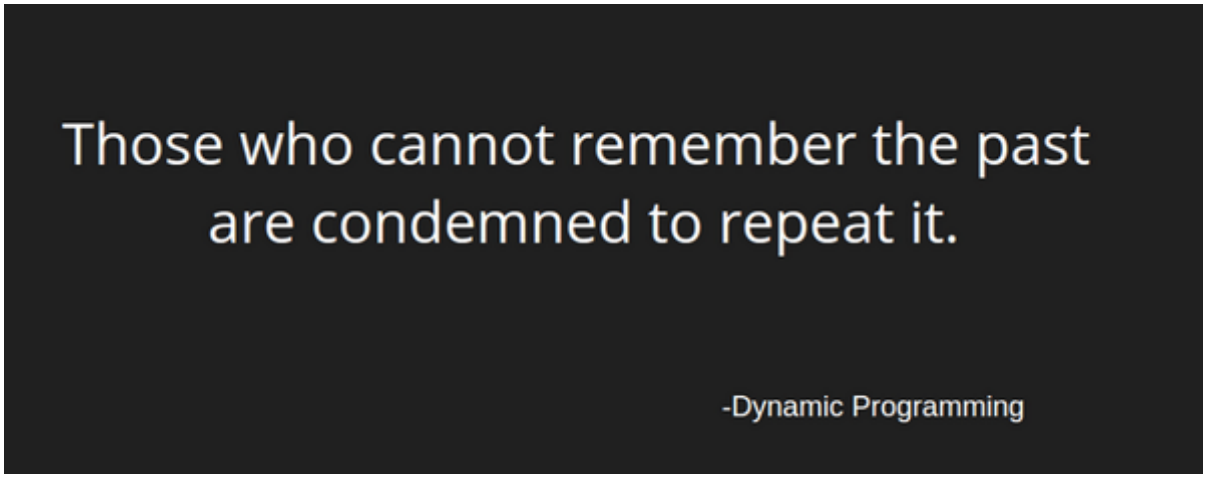
Solve any problem to achieve a rank

View Leaderboard

Topics: Introduction to Dynamic Programming 1

Introduction to Dynamic Programming 1

TUTORIAL PROBLEMS



The image above says a lot about Dynamic Programming. So, is repeating the things for which you already have the answer, a good thing ? A programmer would disagree. That's what Dynamic Programming is about. To *always* remember answers to the sub-problems you've already solved.

Let us say that we have a machine, and to determine its state at time **t**, we have certain quantities called state **variables.** There will be certain times when we have to make a decision which affects the state of the system, which may or may not be known to us in advance. These decisions or changes are equivalent to transformations of state variables. The results of the previous decisions help us in choosing the future ones.

?

What do we conclude from this? We need to break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems. If you are given a problem, which can be broken down into smaller sub-problems, and these smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some over-lappping sub-problems, then you've encountered a DP problem.

Some famous Dynamic Programming algorithms are:

- Unix diff for comparing two files
- Bellman-Ford for shortest path routing in networks
- TeX the ancestor of LaTeX
- WASP - Winning and Score Predictor

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds it application in a lot of real life situations.

In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

Jonathan Paulson explains Dynamic Programming in his amazing Quora answer here.

> Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
> "What's that equal to?"
> Counting "Eight!"
> **Writes down another "1+" on the left.**
> **"What about that?"**
> "Nine!" " How'd you know it was nine so fast?"
> "You just added one more!"
> "So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Dynamic Programming and Recursion:

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

?

Fibonacci (n) = 1; if n = 0
Fibonacci (n) = 1; if n = 1
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: **1, 1, 2, 3, 5, 8, 13, 21...** and so on!

A code for it using pure recursion:

```
 int fib (int n) {
       if (n < 2)
            return 1;
       return fib(n-1) + fib(n-2);
   }
```

Using Dynamic Programming approach with memoization:

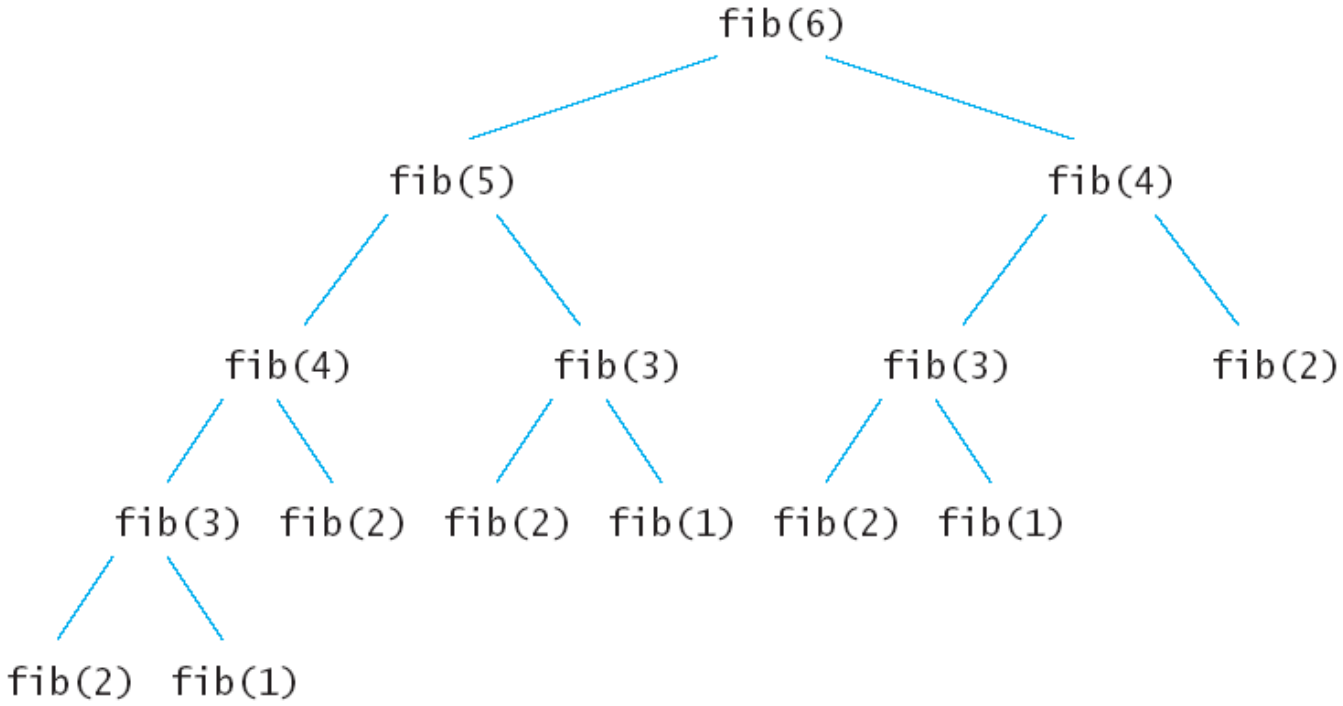
```
void fib () {
       fibresult[0] = 1;
       fibresult[1] = 1;
       for (int i = 2; i<n; i++)
          fibresult[i] = fibresult[i-1] + fibresult[i-2];
   }
```

Are we using a different recurrence relation in the two codes? No. Are we doing anything different in the two codes? Yes.

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once. Take a look at the image to understand that how certain

?

values were being recalculated in the recursive way:



Majority of the Dynamic Programming problems can be categorized into two types:

1. Optimization problems.
2. Combinatorial problems.

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

Bottom up vs. Top Down:

- **Bottom Up** - I'm going to learn programming. Then, I will start practicing. Then, I will start taking part in contests. Then, I'll practice even more and try to improve. After working hard like crazy, I'll be an amazing coder.
- **Top Down** - I will be an amazing coder. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll start taking part in contests. Then? I'll practicing. How? I'm going to learn programming.

Not a great example, but I hope I got my point across. In Top Down, you start building the big solution right away by explaining how you build it from smaller solutions. In Bottom Up, you start

with the small solutions and then build up.

Memoization is very easy to code and might be your first line of approach for a while. Though, with dynamic programming, you don't risk blowing stack space, you end up with lots of liberty of when you can throw calculations away. The downside is that you have to come up with an ordering of a solution which works.

One can think of dynamic programming as a table-filling algorithm: you know the calculations you have to do, so you pick the best order to do them in and ignore the ones you don't have to fill in.

Let's look at a sample problem:

> Let us say that you are given a number **N,** you've to find the number of different ways to write it as the sum of 1, 3 and 4.

For example, if N = 5, the answer would be 6.

- 1 + 1 + 1 + 1 + 1
- 1 + 4
- 4 + 1
- 1 + 1 + 3
- 1 + 3 + 1
- 3 + 1 + 1

Sub-problem: DP_n be the number of ways to write **N** as the sum of 1, 3, and 4.
Finding recurrence: Consider one possible solution, n = x1 + x2 + ... x_n. If the last number is 1, the sum of the remaining numbers should be n - 1. So, number of sums that end with 1 is equal to DP_{n-1}. Take other cases into account where the last number is 3 and 4. The final recurrence would be:

> $DP_n = DP_{n-1} + DP_{n-3} + DP_{n-4}$.

Take care of the base cases. $DP_0 = DP_1 = DP_2 = 1$, and $DP_3 = 2$.

Implementation:

```
DP[0] = DP[1] = DP[2] = 1; DP[3] = 2;
    for (i = 4; i <= n; i++) {
      DP[i] = DP[i-1] + DP[i-3] + DP[i-4];
    }
```

The technique above, takes a bottom up approach and uses memoization to not compute results that have already been computed.

I also want to share Michal's amazing answer on Dynamic Programming from Quora.

> "Imagine you have a collection of **N** wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from **1 to N**, respectively. The price of the i^{th} wine is pi. (prices of different wines can be different).
>
> Because the wines get better every year, supposing today is the year 1, on year y the price of the i^{th} wine will be y*pi, i.e. y-times the value that current year.
>
> You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).
>
> You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order?"

So, for example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3. The optimal solution would be to sell the wines in the order **p1, p4, p3, p2** for a total profit 1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29.

Wrong solution!

After playing with the problem for a while, you'll probably get the feeling, that in the optimal solution you want to sell the expensive wines as late as possible. You can probably come up with the following greedy strategy:

> Every year, sell the cheaper of the two (leftmost and rightmost) available wines.

Although the strategy doesn't mention what to do when the two wines cost the same, this strategy feels right. But unfortunately, it isn't, as the following example demonstrates.

If the prices of the wines are: p1=2, p2=3, p3=5, p4=1, p5=4. The greedy strategy would sell them in the order p1, p2, p5, p4, p3 for a total profit 2 * 1 + 3 * 2 + 4 * 3 + 1 * 4 + 5 * 5 = 49.

But, we can do better if we sell the wines in the order **p1, p5, p4, p2, p3** for a total profit 2 * 1 + 4 * 2 + 1 * 3 + 3 * 4 + 5 * 5 = 50.

This counter-example should convince you, that the problem is not so easy as it can look on a first sight and it can be solved using DP.

How? Write a backtrack.

When coming up with the memoization solution for a problem, start with a backtrack solution that finds the correct answer. Backtrack solution enumerates all the valid answers for the problem and chooses the best one.

Here are some restrictions on the backtrack solution:

- It should be a function, calculating the answer using recursion.

- It should return the answer with return statement, i.e., not store it somewhere.
- All the non-local variables that the function uses should be used as read-only, i.e. the function can modify only local variables and its arguments.

```
int p[N]; // read-only array of wine prices
// year represents the current year (starts with 1)
// [be, en] represents the interval of the unsold wines on the shelf
    int profit(int year, int be, int en) {
      // there are no more wines on the shelf
      if (be > en)
        return 0;

     // try to sell the leftmost or the rightmost wine, recursively calculate the
      // answer and return the better one
        return max(
        profit(year+1, be+1, en) + year * p[be],
        profit(year+1, be, en-1) + year * p[en]);
}
```

This solution simply tries all the possible valid orders of selling the wines. If there are **N** wines in the beginning, it will try 2^N possibilities (each year we have 2 choices). So even though now we get the correct answer, the time complexity of the algorithm grows exponentially.

The correctly written backtrack function should always represent an answer to a well-stated question. In our case profit function represents an answer to a question: "*What is the best profit we can get from selling the wines with prices stored in the array p, when the current year is year and the interval of unsold wines spans through [be, en], inclusive?*"

You should always try to create such a question for your backtrack function to see if you got it right and understand exactly what it does.

We should try to minimize the state space of function arguments. In this step think about, which of the arguments you pass to the function are redundant. Either we can construct them from the other arguments or we don't need them at all. If there are any such arguments, don't pass them to the function. Just calculate them inside the function.

In the above function profit, the argument year is redundant. It is equivalent to the number of wines we have already sold plus one, which is equivalent to the total number of wines from the beginning minus the number of wines we have not sold plus one. If we create a read-only **global variable N,** representing the total number of wines in the beginning, we can rewrite our function as follows:

```
int N; // read-only number of wines in the beginning
   int p[N]; // read-only array of wine prices

       int profit(int be, int en) {
         if (be > en)
           return 0;
```

```
        // (en-be+1) is the number of unsold wines
        int year = N - (en-be+1) + 1; // as in the description above
         return max(
          profit(be+1, en) + year * p[be],
          profit(be, en-1) + year * p[en]);
      }
```

We are now 99% done. To transform the backtrack function with time complexity $O(2^N)$ into the memoization solution with time complexity $O(N^2)$, we will use a little trick which doesn't require almost any thinking. As noted above, there are only $O(N^2)$ different arguments our function can be called with. In other words, there are only $O(N^2)$ different things we can actually compute.

So where does $O(2^N)$ time complexity comes from and what does it compute? The answer is - the exponential time complexity comes from the repeated recursion and because of that, it computes the same values again and again. If you run the above code for an arbitrary array of N=20 wines and calculate how many times was the function called for arguments be=10 and en=10 you will get a number 92378.

That's a huge waste of time to compute the same answer that many times. What we can do to improve this is to memoize the values once we have computed them and every time the function asks for an already memoized value, we don't need to run the whole recursion again.

```
int N; // read-only number of wines in the beginning
    int p[N]; // read-only array of wine prices
    int cache[N][N]; // all values initialized to -1 (or anything you
choose)

      int profit(int be, int en) {
        if (be > en)
          return 0;
       // these two lines save the day
        if (cache[be][en] != -1)
          return cache[be][en];
        int year = N - (en-be+1) + 1;
        // when calculating the new answer, don't forget to cache it
        return cache[be][en] = max(
          profit(be+1, en) + year * p[be],
          profit(be, en-1) + year * p[en]);
      }
```

To sum it up, if you identify that a problem can be solved using DP, try to create a backtrack function that calculates the correct answer. Try to avoid the redundant arguments, minimize the range of possible values of function arguments and also try to optimize the time complexity of one functior

call (remember, you can treat recursive calls as they would run in O(1) time). Finally, you can memoize the values and don't calculate the same things twice.

Read Michal's another cool answer on Dynamic Programming here.

Contributed by: Prateek Garg