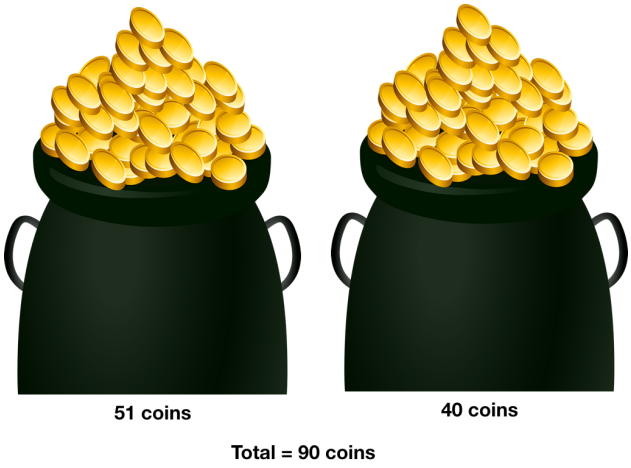
Dynamic Programming

Dynamic programming is basically an optimization algorithm. It means that we can solve any problem without using dynamic programming but we can solve it in a better way or optimize it using dynamic programming.

Idea Behind Dynamic Programming

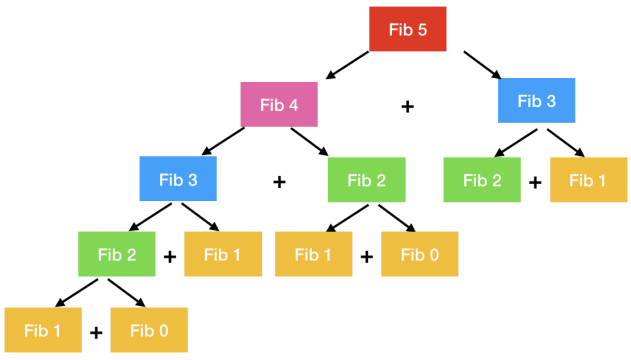
The basic idea of dynamic programming is to store the result of a problem after solving it. So when we get the need to use the solution of the problem, then we don't have to solve the problem again and just use the stored solution.

Imagine you are given a box of coins and you have to count the total number of coins in it. Once you have done this, you are provided with another box and now you have to calculate the total number of coins in both boxes. Obviously, you are not going to count the number of coins in the first box again. Instead, you would just count the total number of coins in the second box and add it to the number of coins in the first box you have already counted and stored in your mind. This is the exact idea behind dynamic programming.



Recording the result of a problem is only going to be helpful when we are going to use the result later i.e., the problem appears again. This means that dynamic programming is useful when a problem breaks into subproblems, the same subproblem appears more than once.

Take a case of calculation of Fibonacci series using recursion i.e., $F(n) = F(n-1) + F(n-2)$ and $F(0) = 0$, $F(1) = 1$.



You can see here that to calculate the 5^{th} term, the same subproblem appears more than once. For example, $F(3)$ is occurring twice, $F(1)$ is occurring 4 times, etc.

So, despite calculating the result of the same problem, again and again, we can store the result once and use it again and again whenever needed.

Let's take look at the code of Fibonacci series without recording the results of the subproblems.

```
FIBONACCI(n)
  if n==0
    return 0
  if n==1
    return 1
  return FIBONACCI(n-1) + FIBONACCI(n-2)
```

C Python Java

```c
#include <stdio.h>

int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }

  if (n == 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
  printf("%d\n",fibonacci(46));
  return 0;
}
```

The code is simple. Since $F(0)$ and $F(1)$ are 0 and 1 respectively, we are handling those cases first. Otherwise, we are calculating the n^{th} term is `FIBONACCI(n-1) + FIBONACCI(n-2)` and we are returning that.

Running this code to calculate the 46^{th} term of the series took around 13 seconds on my computer in C.

Using Dynamic Programming

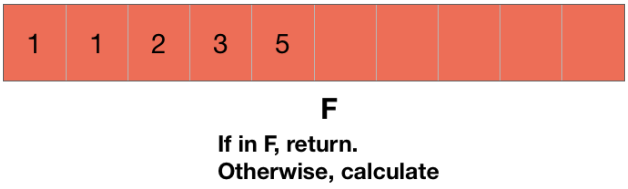
Let's write the same code but this time by storing the terms we have already calculated.

```
F = [] //new array
FIBONACCI(n)
  if F[n] == null
    if n==0
      F[n] = 0
    else if n==1
      F[n] = 1
    else
      F[n] = FIBONACCI(n-1) + FIBONACCI(n-2)
  return F[n]
```

C Python Java

```
#include <stdio.h>

int F[50]; //array to store fibonacci terms

// Initially, all elements of array F are -1.
void init_F() {
  int i;
  for(i=0; i<50; i++) {
    F[i] = -1;
  }
}

int dynamic_fibonacci(int n) {
  if (F[n] < 0) {
    if (n==0) {
      F[n] = 0;
    }
    else if (n == 1) {
      F[n] = 1;
    }
    else {
      F[n] = dynamic_fibonacci(n-1) + dynamic_fibonacci(n-2);
    }
  }
  return F[n];
}

int main() {
  init_F();
  printf("%d\n",dynamic_fibonacci(46));
  return 0;
}
```

Here, we are first checking if the result is already present in the array or not `if F[n] == null`. If it is not, then we are calculating the result and then storing it in the array F and then returning it `return F[n]`.



Running this code for the 100^{th} term gave the result almost instantaneously and this is the power of dynamic programming.

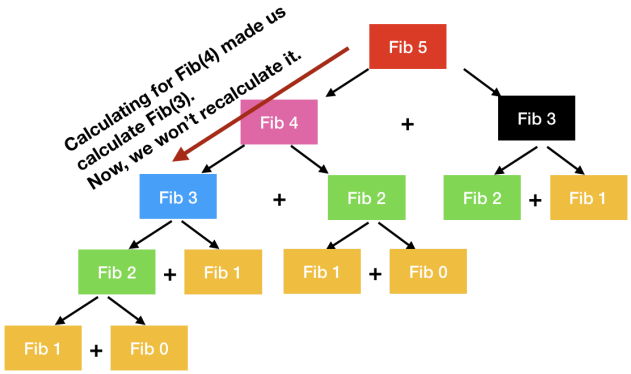
But are we sacrificing anything for the speed? Yes, memory. Dynamic programming basically trades time with memory. Thus, we should take care that not an excessive amount of memory is used while storing the solutions.

Two Approaches of Dynamic Programming

There are two approaches of the dynamic programming. The first one is the top-down approach and the second is the bottom-up approach. Let's take a closer look at both the approaches.

Top-Down Approach

The way we solved the Fibonacci series was the top-down approach. We just start by solving the problem in a natural manner and stored the solutions of the subproblems along the way. We also use the term **memoization**, a word derived from memo for this.



In other terms, it can also be said that we just hit the problem in a natural manner and hope that the solutions for the subproblem are already calculated and if they are not calculated, then we calculate them on the way.

Bottom-Up Approach

The other way we could have solved the Fibonacci problem was by starting from the bottom i.e., start by calculating the 2^{nd} term and then 3^{rd} and so on and finally calculating the higher terms on the top of these i.e., by using these values.

F(0) = 1
F(1) = 1

Fib 2 **Calculate F(2)**

Fib 3 **Then calculate F(3)**

And so on ...

We use a term **tabulation** for this process because it is like filling up a table from the start.

Let's again write the code for the Fibonacci series using bottom-up approach.

```
F = [] //new array
FIBONACCI-B-UP(n)
  F[0] = 0
  F[1] = 1

  for i in 2 to n
    F[i] = F[i-1] + F[i-2]

  return F[n]
```

C Python Java

```
#include <stdio.h>

int F[50]; //array to store fibonacci terms

int fibonacci_bottom_up(int n) {
  F[n] = 0;
  F[1] = 1;

  int i;
  for(i=2; i<=n; i++) {
    F[i] = F[i-1] + F[i-2];
  }
  return F[n];
}

int main() {
  printf("%d\n",fibonacci_bottom_up(46));
  return 0;
}
```

As said, we started calculating the Fibonacci terms from the starting and ended up using them to get the higher terms.

Also, the order for solving the problem can be flexible with the need of the problem and is not fixed. So, we can solve the problem in any needed order.

Generally, we need to solve the problem with the smallest size first. So, we start by sorting the elements with size and then solve them in that order.

Let's compare memoization and tabulation and see the pros and cons of both.

Memoization V/S Tabulation

Memoization is indeed the natural way of solving a problem, so coding is easier in memoization when we deal with a complex problem. Coming up with a specific order while dealing with lot of conditions might be difficult in the tabulation.

Also think about a case when we don't need to find the solutions of all the subproblems. In that case, we would prefer to use the memoization instead.

However, when a lot of recursive calls are required, memoization may cause memory problems because it might have stacked the recursive calls to find the solution of the deeper recursive call but we won't deal with this problem in tabulation.

Generally, memoization is also slower than tabulation because of the large recursive calls.

Thus, we have seen the idea, concepts and working of dynamic programming in this chapter. We are going to discuss some common algorithms using dynamic programming.

Also, you can share your knowledge with the world by writing an article about it on BlogsDope (https://www.codesdope.com/blog/submit-article/).

> “ I learned the value of hard work by working hard. ”
>
> - Margaret Mead

PREV (/course/algorithms-binary-search/) (/course/algorithms-knapsack-problem/) NEXT