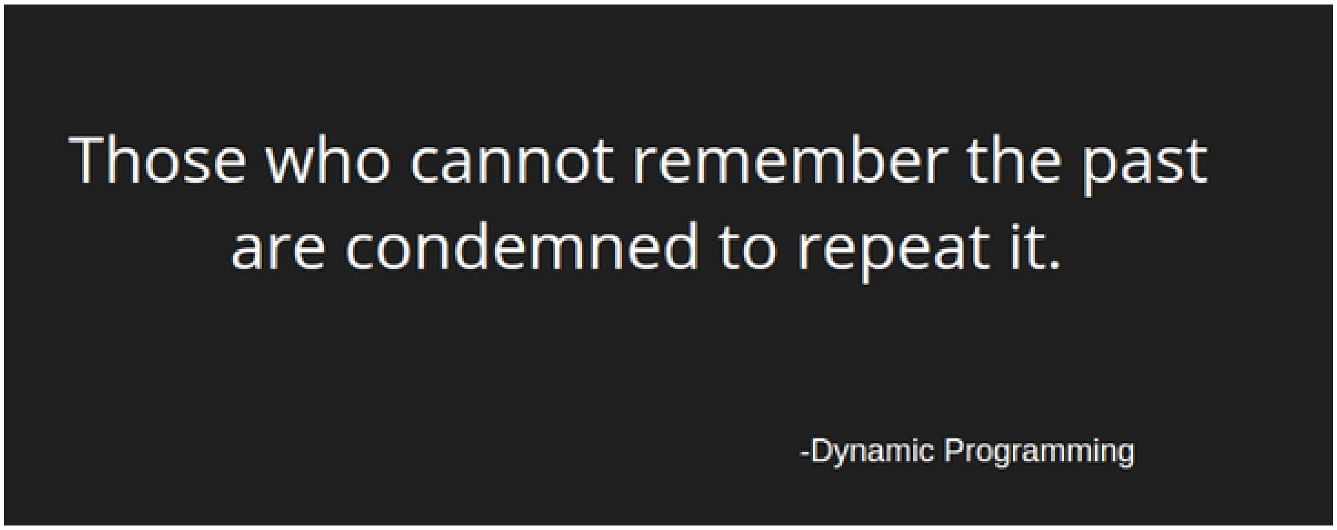
Nazir Nayal-Moayed Haji Ali
Koc University

Dynamic Programming 1

Algorithms Program
February 27, 2020

Contents:



Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

Figure 1: A Wise Saying

Intro:

Definition

Dynamic Programming (DP for short) is a technique used to solve optimization or counting problems. In 85% of the cases you will use DP to:

- **Maximize**
- **Minimize**
- **Count**

For programming competitions or technical interviews, this technique is one of the most commonly used and it can be used to solve from easy to extremely difficult questions.

There are multiple ways to solve a problem using DP, we will see several examples to learn about each.

But generally speaking, DP can be used to optimize some brute force solutions of some problems that have a property called **optimal substructure**, by sacrificing **memory** to increase **speed**.

Optimal Substructure:

We say that a problem has an **Optimal Substructure** when we are able to compute the optimal solution of the problem by using the optimal solutions of smaller versions of the problem. A very elegant representation of this concept would be **recurrence relations**, which are functions that usually written in terms of the same function with smaller input values. **In Fact**, recurrence relations are the main representation used to define the DP sub-problems and build DP solutions.

A simple example:

An example to understand how optimal substructure can be utilized be the following.

If someone asks you to calculate the answer of 713×17, it might take you around a minute to find the solution using a paper and a pen (assuming your mental math skills are average). But if before hand I told you that $712 \times 17 = 12104$ it It should take you much less time to find the answer, because all you need to do it add 17 to the answer I gave you. So you were able to take advantage of a smaller version of the multiplication to **quickly** find the answer for the bigger one. This is one way only to think about how DP works.

Now let's look at several simple problems to build a better understanding of DP.

Finding Fibonacci Numbers with DP:

Fibonacci Numbers form a very famous series, where the first two numbers are assumed to be equal to 0 and 1 by default, and each number following in the series is the sum of the two previous numbers. This is represented using the following recurrence relation:

$$F(N) = F(N-1) + F(N-2)$$

$$BaseCase : F(0) = 0\ ,\ F(1) = 1$$

Mathematically speaking, the i^{th} Fibonacci number can be calculated using a closed formula, which in programming is equivalent to $O(1)$ complexity. The formula is as follows:

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

So why not use the formula which is very fast and instead look for another solution ? Good question, the reason is that this formula contains division and square root operations, and when computing it using a computer, we are bound to precision errors, even though we are expecting integer outputs. Hence, it would be much safer and accurate to compute Fibonacci numbers using another means. So let's see how to calculate it using the recurrence formula.

Brute Force Approach:

Normally, implementing recurrent relations can be done using recursive functions, because of the great analogy. So we will implement this using a function also called $F(n)$. Observe the code and how recursion enables us to elegantly implement recurrence relations.

Algorithm 1: Fibonacci Numbers: Brute Force

```
#include <bits/stdc++.h>

using namespace std;

int F(int n){
  // first see if we reached the base cases
  if(n == 0) return 0;
  if(n == 1) return 1;
  // otherwise simply return the sum of the samller
  // subproblems.
  return F(n - 1) + F(n - 2);
}

int main(){

  int n = 6;

  cout << F(n) << endl;

}
```

Now let's try to find the complexity of this algorithm. Basically the cost we are doing here is the number of calls for the function F, so in order to estimate how many calls we are making in order to compute $F(n)$, we use a tree representation to find a pattern that would allow us to count the number of calls. Observe the following tree.

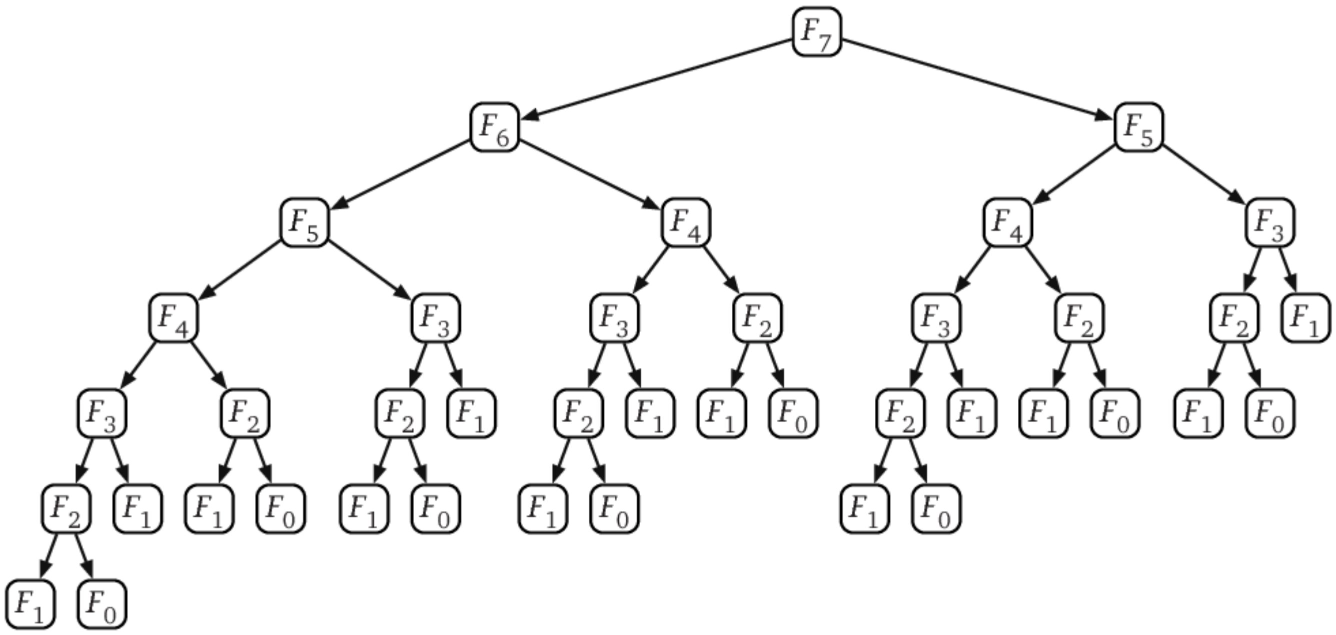


Figure 2: Fibonacci Recursion Tree [source included below]

Each node in the tree corresponds to a single call for $F(n)$. Notice that from every node, there are two nodes resulting in the next level, so each level is essentially double the number of nodes of the previous level, which implies approximately $O(2^n)$ complexity.

So in order to calculate $F(7)$, there are around 41 calls!. This happens because if you take a closer look at the calls you will see that calls for a certain value are being recalculated more than once. $F(3)$ for example is being called 5 times. $F(2)$ called 8 times!

There is an apparent waste of resources to calculate the same thing over again. So in order to optimize the solution we can use Dynamic Programming.

Dynamic Programming Approach:

In order to waste the time calculating thing over and over again, we will store the values we compute, so that when we need them in the future for some other recursive call, we can simply use the value store instead of going down the tree once again.

A simple modification to our previous code will make the calculations much faster. We only only define an array $Memo[n]$ that will work as a storage table, we give it the index (order) of the Fibonacci number we want and it will give us the number. Here is the code:

Algorithm 2: Fibonacci Numbers: Dynamic Programming

```
#include <bits/stdc++.h>

using namespace std;
```

```
const int MaxN = 1e6;

int Memo[MaxN];

int F(int n){
  // first see if we reached the base cases
  if(n == 0) return 0;
  if(n == 1) return 1;

  // If Memo[n] doesn't equal -1, this means it has been
  // calculated before so just return the stored value
  if(Memo[n] != -1) return Memo[n];

  // otherwise calculate the subproblems and
  // store them in the Memo table.
  Memo[n] = F(n - 1) + F(n - 2);
  return Memo[n];
}

int main(){

  // we fill all elements of Memo in the beginning with
  // values -1 to indicate that its value hasn't been
  // calculated yet
  memset(Memo,-1,sizeof(Memo));

  int n = 6;

  cout << F(n) << endl;

}
```

With this optimization, now every $F(n)$ needs to be calculated only once. We used some memory to gain speed. The complexity of this approach becomes $O(n)$, because this is the maximum number of calls we need to make to find the n^{th} Fiboncci number.

Understanding DP states:

Before proceeding to more DP problems, we will introduce a very important term that is usually used to define DP problems, it is **Dp states**.

DP states are the variables that define a sub-problem; these are the values which help us distinguish between sub-problems, and know that this sub-problem is different than the other. They basically represent an *ID* for each sub-problem. In Fibonacci Algorithm, the state is only one variable, which is n, because we know that for a fixed value of n, we are calculating the same sub-problem; we know that if we have the value of $F(n)$, we do not need to calculate it again, but a different value of n defines a different sub-problem which needs to be calculated separately.

This concept might not be fully clear from the above explanation, but as we introduce problems we will define their states and hopefully the reader would get a better idea of the concept.

The concept of state is important because once you know the states of the DP problem you will be able to easily define a recurrence relation and implement the solution. Also, the states are usually the things that we store in the memorization table, because as mentioned above, the uniquely define each sub-problem.

Minimize coins:

Problem Definition:

Given a set of n coin values $S\{c_1, c_2 \dots c_n\}$, we need to form a target value $target$ using a **minimum** number of coins. Assume that you have an infinite amount of each coin type. The task is to print the minimum amount of coins to be used. Let's see how we can approach this problem.

Greedy Approach ?

First, let's check if a greedy approach works to solve this problem. The obvious greedy idea would be to keep choosing the greatest possible coin every time until we reach the sum we need. This approach does **not** produce optimal answer.

Observe this counter example: Assume we have the coins $\{1, 3, 4\}$, and we want to form the sum 10. Following the greedy approach, we will take $\{4, 4, 1, 1\}$, which gives us a result of 4. This is not the optimal solution since we can achieve 3 by choosing $\{3, 3, 4\}$.

Brute Force Approach:

Let's see now how to solve it using Brute Force. We will try all possible combinations of coins, and we will choose the one with minimum number of coins involved. This approach is definitely correct because it attempts all the possible solutions. But how to implement it ? Let's try to create a recurrence relation that is similar to the Fibonacci Problem above.

Assume we have have a function $solve(x)$ similar to $F(n)$. This function computes the minimum number of coins required to form a sum x from a fixed set of coins.

How can $solve(x)$ depend on smaller versions of the problem like in Fibonacci Numbers ? Let's assume that our coins are $\{1, 3, 4\}$. Observe that the only way to reach a sum x, is by first obtaining one of these sums: $\{x - 1, x - 3, x - 4\}$, and adding a coin to it which will give us x. Hence we can say that the minimum cost of reaching x is the the minimum cost of reaching any of the three sums $\{x-1, x-3, x-4\}$, plus 1 for the additional coin we are adding to reach x. Hence we can define the following recurrence relation:

$$solve(x) = 1 + min(solve(x-1), solve(x-2), solve(x-4))$$

That's nice, now we have a recurrence relation which can be implemented using a recursive function. But this is a very specific example, so in order to generalize it for any set of coins, we can simply iterate over all available coins $\{c_1, c_2, \ldots, c_n\}$ and call $solve(x - c_i)$ on each and choose the one that produces the minimum answer.

Also, what is the base case for this recurrence relation ? In Fibonacci numbers we said that whenever we reach 0 or 1, we return default values we have defined. For this problem, observe that not all coin sequences can give a sum x, but the sequences that do will reach $solve(0)$, because when you keep subtracting x with a sequence of coins and you reach 0, this means that there exists a valid sequence of coins. If a sequence of choices leads to a negative x, this it is not valid, hence we can terminate by signaling an invalid sequence. Here is the implementation:

Algorithm 3: Coin Problem: Brute Force

```
#include <bits/stdc++.h>
using namespace std;
const int MaxN = 1e6;
const int Inf = 1e9;

vector<int> Coins;

int solve(int x){

  // if we get a negative sum return Infinity
  // which is just a very big number so that the choice
  // that has lead to this sum is rejected
  // (it will be rejected because we are taking minimum)
  if(x <  0) return Inf;

  // if we reach a sum 0, then return 0 to accept this solution
  if(x == 0) return 0;

  // initialize the answer for this sub-problem as infinity
  // in order to make sure all possible solutions are considered
  int Ans = Inf;

  // iterate over all coins
  for(auto c : Coins){
    // solve(x - c) is solving the smaller subproblem
    // for every c in coins
    Ans = min(Ans,1 + solve(x - c));
  }
  // return the minimum answer found this subproblem.
  return Ans;
}

int main(){
  Coins.push_back(1);
  Coins.push_back(3);
  Coins.push_back(4);
  int target = 10;
  cout << solve(target);
}
```

Now let's consider the cost of this approach. With a similar simple analysis to the Fibonacci problem, it can be proven that the complexity of this approach is exponential. Try to draw a tree and count the number of levels and how the number of nodes increase from one level to another. You'll find that the complexity is around $O(n^x)$. Again Let's use DP to optimize the solution.

Dynamic Programming Approach:

If you draw the tree, again, you will notice that many calls for the same x are being done, so let's store them in the table $Memo[x]$. Here is the code:

Algorithm 4: Coin Problem: Dynamic Programming

```
#include <bits/stdc++.h>
using namespace std;
const int MaxN = 1e6;
const int Inf = 1e9;

vector<int> Coins;

int Memo[MaxN];

int solve(int x){

  // if we get a negative sum return Infinity
  // which is just a very big number so that the choice
  // that has lead to this sum is rejected
  // (it will be rejected because we are taking minimum)
  if(x <  0) return Inf;

  // if we reach a sum 0, then return 0 to accept this solution
  if(x == 0) return 0;

  // if Memo[x] != -1, then the solution is already computed
  if(Memo[x] != -1) return Memo[x];

  // initialize the answer for this sub-problem as infinity
  // in order to make sure all possible solutions are considered
  int Ans = Inf;

  // iterate over all coins
  for(auto c : Coins){
    // solve(x - c) is solving the smaller subproblem
    // for every c in coins
    Ans = min(Ans,1 + solve(x - c));
  }

  // We save the answer in our table
  Memo[x] = Ans;

  // return the minimum answer found this subproblem.
  return Ans;
}

int main(){

  // initialize all values of Memo to -1 to signal
  // that the solution for some Memo[x] hasn't been calculated
  memset(Memo,-1,sizeof(Memo));

  Coins.push_back(1);
  Coins.push_back(3);
  Coins.push_back(4);
  int target = 10;
  cout << solve(target);
}
```

The complexity now drops to $O(n \times x)$, because we are only making x calls to the function, and in each call we are iterating over all coins (remember there were n coins).

So what are the DP states in this problem ? The state here is x, which is the required target sum. Observe that it uniquely identifies each sub-problem we might face.

It is quite interesting that using DP, we are able to transform a slow brute force solution to an efficient one by adding a few lines of code!

There is another DP way to approach this coin problem, which is included in the end of the document. This proves that there might be several ways to define recurrence relations and solve DP problems. The next section introduces two distinct paradigms to solve DP problems.

Top-Down vs Bottom-Up DP

There are two very distinctive way to solve DP problems, called **Top-Down** and **Bottom-Up** DP. All the examples we have seen so far are classified as Top-Down. The difference between the two approaches is the way we are using the sub-problems and from which direction we are proceeding as we look for the solution.

Top-Down DP:

In Top-Down, when we solved Fibonacci problem, here is how we were thinking:

Okay, I want to find $F(n)$, so I need the values of $F(n-1)$ and $F(n-2)$, okay let's go find them and then compute the solution for $F(n)$ and we are done.

As you can see, we are kind of descending from the solution we want, finding answers for sub-problems we need, and using those to build the big solution.

Bottom-Up DP:

This method build the solution in a different way. Instead of starting from the top, and seek values from below, we start from the bottom, which is represented by the base case, and build the solution for all possible states as we go up, until we reach our desired state.

Bottom-Up, unlike Top-Down, does not use the recursive function property. It is usually implemented using for loops. Here are below the Bottom-Up versions of Fibonacci and Coin problem:

Algorithm 5: Fibonacci: Bottom-Up DP

```
#include <bits/stdc++.h>
using namespace std;
const int MaxN = 1e6;
const int Inf = 1e9;

vector<int> Coins;

int Memo[MaxN];

int main(){
  memset(Memo,0,sizeof(Memo));

  int n = 6;
  Memo[0] = 0;
  Memo[1] = 1;

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

  cout << Memo[n] << endl;

}
```

Algorithm 6: Coin Problem: Bottom-Up DP

```
#include <bits/stdc++.h>
using namespace std;
const int MaxN = 1e6;
const int Inf = 1e9;

vector<int> Coins;

int Memo[MaxN];

int main(){

  Coins.push_back(1);
  Coins.push_back(3);
  Coins.push_back(4);
  int target = 10;


  for(int i = 0 ; i <= target ; ++i)
      Memo[i] = Inf;

  Memo[0] = 0;

  for(int x = 1 ; x <= target ; ++x){
    for(auto c : Coins){
      if(x - c >= 0){
        Memo[x] = min(Memo[x],1 + Memo[x - c]);
      }
    }
  }

  cout << Memo[target] << endl;

}
```

Which is Better ?

Each of the two approaches has its ups and downs. Here are some notable distinctions:

- Bottom-up will visit all possible states regardless of whether we need to calculate it or not, whereas Top-Down will only visit and calculate the state we need, which can be more efficient sometimes.
- Bottom up enables us to do some memory optimizations. Observe that in Fibonacci case, all we need to store while building the solution is the last 2 numbers used. Hence

we will not need to maintain an $O(n)$ table, It's enough to store the last two integers used only.

Appendix:

Another DP Approach for Coin Problem:

Back-Tracking is a recursive approach to solve problems that involve making a sequences of choices, it uses the recursive property of functions to try every possible sequence of choices, and choose the one that we think is best.

In our case, we need to check a sequence of coins, so in order to simplify understanding this approach, say we have a function, let's call it $solve()$, that accepts only 1 variable, which is the sum of the values of coins we have collected so far. Let's call it sum. $solve(sum)$'s job is to take sum, and try to add the value of each of the coins that we have to the sum, and send this sum to another call of the same function to do the same computations. Here are the computations made by $solve(sum)$:

- $solve(sum)$ knows that if sum is greater than the target, then continuing is pointless, and the sequence of choices that have been made before reaching the function cannot lead to the target, hence it will return a signal to ignore this sequence.
- When sum is equal $target$, this means this is a valid sequence, so it returns a signal that the sequence leading to this sum is valid, and no further additions are required.
- The last case is when sum less than $target$, this means we will need to use at least one of the coins we have, which also means that the solution of the sequence that has led us to sum must increase by 1. So if we try each single coin, the best choice would be the one that ends first (returns minimum number of additional coins), so iterate over the coins we have and we do the recursive call on the function with $solve(sum + a_i)$. Each function will return the number of calls that have been done after it to reach an answer, or Inf. So we take the minimum of all these cases and we add 1 to it (which is the cost of taking a coin). Then we return this value, which if you closely think of it, it corresponds to the minimum number of coins we need to take to go from sum to $target$.

Here is a commented code that it the implementation of the described approach above.

Algorithm 7: Coin Problem: Back Tracking

```
#include <bits/stdc++.h>
using namespace std;
```

```
const int Inf = 1e9;

vector<int> Coins;
int target;
// sum is the sum we accumulated so far
int solve(int sum){
  if(sum > target) return Inf; // infinity means invalid sequence
  if(sum == target) return 0; // 0 means a valid sequence
  // Inf is a very big value, since we want the minimum then in order
  // to spot the minimum we need to start a number bigger than all
  // possible answers we could get
  int Ans = Inf;
  // iterate over all coins and try to add each one
  for(int c : Coins){
    // we take minimum of answer we have so far, and the answer
    // resulting from adding the coin $coin[i]$
     Ans = min(Ans,1 + solve(sum + c));
  }
  // this answer counts the minimum possible coins to use
  // starting from the sum we got so final answer should be
  // solve(0)
  return Ans;
}

int main(){
  Coins.push_back(1);
  Coins.push_back(3);
  Coins.push_back(4);
  target = 10;
  cout << solve(0) << endl;
}
```

The problem of the previous approach is that it is very slow, because trying all possible sequences to reach a sum D can cost up to n^D on average case. Detailed analysis will be omitted. Also, like in the Fibonacci problem, we are subject to calculating some values more than once. For example during the search a certain number X might be reached using several sequences. Take $\{1, 3, 4\}$, in our algorithm, we will call $solve(9)$ several times using the following sequences :

- $\{1, 4, 4\}$
- $\{1, 1, 3, 4\}$
- $\{1, 1, 1, 3, 3\}$
- $\{1, 1, 1, 1, 1, 4\}$
- $\{1, 1, 1, 1, 1, 1, 1, 3\}$

- $\{1,1,1,1,1,1,1,1,1\}$

So for a very small value 9, we will make the same call 5 times, imagine how many identical calls we are making for bigger numbers. This is a very huge waste of time, so DP comes to save the day.

As we said earlier, Dynamic Programming is about storing repeated calculations to make them reusable and save time calculating them. So in order to optimize our solution above, we need a table or an array, which will be used to store each function call answer, so that whenever a call is done for any already calculated *sum*, we simply return the value we have calculated it instead of making new recursive calls.

With a slight modification of our previous code we get an optimized solution:

Algorithm 8: Coin Problem: Dynamic Programming

```
#include <bits/stdc++.h>
using namespace std;

const int Inf = 1e9;
const int MaxSum = 1e6;

vector<int> Coins;
int target;

// This is the DP table we will use to store the computed values
int DP[MaxSum];

// sum is the sum we accumulated so far
int solve(int sum){
  if(sum > target) return Inf; // infinity means invalid sequence
  if(sum == target) return 0; // 0 means a valid sequence

  // If DP[sum] isn't -1, then we have previously
  // calculated solve(sum), hence we just return the computed value
  if(DP[sum] != -1) return DP[sum];

  // Inf is a very big value, since we want the minimum then in order
  // to spot the minimum we need to start a number bigger than all
  // possible answers we could get
  int Ans = Inf;
  // iterate over all coins and try to add each one
  for(int c : Coins){
    // we take minimum of answer we have so far, and the answer
    // resulting from adding the coin $coin[i]$
     Ans = min(Ans,1 + solve(sum + c));
  }

  // We store the answer we got for the call solve(sum) in DP[sum]
  DP[sum] = Ans;
  return Ans;
}

int main(){


  // We initialize all values of the DP table to -1
  // so that we know that if it's value is -1, then
  // its value hasn't been computed yet.

  memset(DP,-1,sizeof(DP));

  Coins.push_back(1);
  Coins.push_back(3);
  Coins.push_back(4);
  target = 10;
  cout << solve(0) << endl;
}
```

Observe that the time complexity of this approach is $O(n \times target)$. The reasoning is as follows: How many times is the function being called ? And how much work are we doing per call ?

Observe that in one call we only do $O(n)$, which comes from iterating over all the coins. Also, we are calling the function at most $O(target)$ times, because several calls to the same value are being stored, hence each call for all possible values in the range $[1, target]$ is done only once. So we end up with $O(n \times target)$

References:

- Borras, J.A. CHAPTER 3 Dynamic Programming 3 . 1 Fibonacci Numbers Recursive Definitions Are Recursive Algorithms.
- Comptitive Programming Handbook by Antti Laaksonen.
- The Algorithm Design Manual by Steven S. Skiena.
- HackerEarth DP Tutorial
- A Very Cool Answer on Quora
- TopCoder DP Tutorial
- Codeforces DP Tutorial