Lecture 25: Dynamic Programming

Date: 11/14/2023

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

- Write down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
- "What's that equal to?"
- Counting "Eight!"
- Write 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** is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems using recursion and storing the results of subproblems to avoid computing the same results again.

⇒ There are two types of DP solutions,
- Top Down (Memoization)
- Bottom Up (Tabulation)

Steps to Solve DP Problems

⇒ **First** step is to figure out a mathematical equation that can give the solution. for example,
- to add two numbers, answer = num1 + num2

- to find factorial of a number, factorial = number * number-1 * ... 1

⇒ **Second** step is to write a recursive function that basically implements the solution.

⇒ **Third** step is to remove redundant/duplicate recursive calls by using memoization technique. (top-down dp solution)

⇒ **Fourth** step is to change the order of problem solving into reverse order. So that we can convert recursive implementation to iterative.

Fibonacci Number / Climbing Stairs

⇒ Given a number n, return n'th fibonacci number.
Example:

- Example 1,
  - Input: n=0
  - Output: 0
- Example 2,
  - Input: n=1
  - Output: 1
- Example 3,
  - Input: n=5
  - Output: 3
- Example 4,
  - Input: n=9
  - Output: 34

Equation

F(n) = F(n-1) + F(n-2)

⇒ There is no proper method to figure out the equation. You just need to do more practice problems.

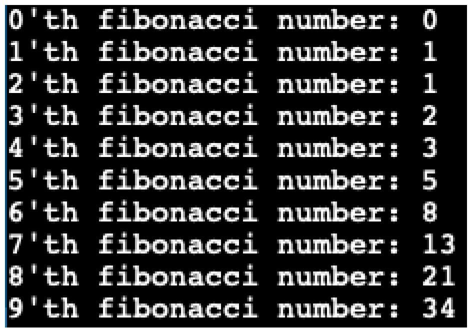
Recursive

⇒ Most of the time you just have to implement the equation created earlier.

```
int F(int n){
      // base cases
      if(n==0)
      return 0;
      if(n==1)
      return 1;

      // equation
      int answer = F(n-1) + F(n-2);

      // return answer
      return answer;
}
```

```
int main(){
      for(int ith=0; ith<10; ith++){
            cout << ith << "'th fibonacci number: " << F(ith) << "\n";
      }

      return 0;
}
```

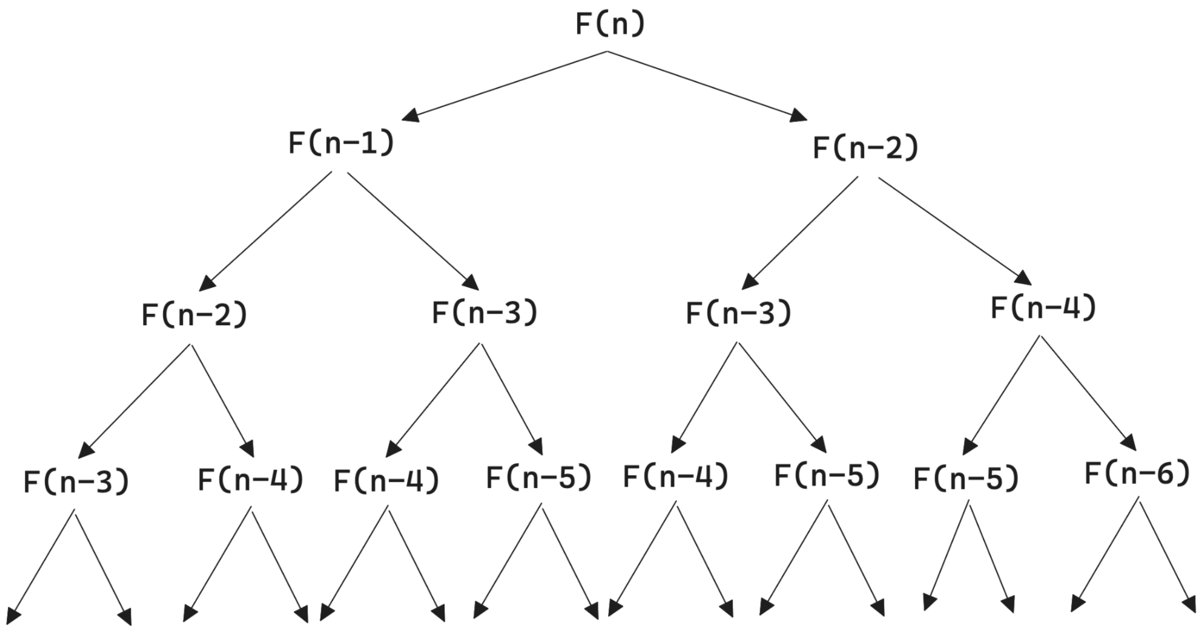
Output:



```
0'th fibonacci number: 0
1'th fibonacci number: 1
2'th fibonacci number: 1
3'th fibonacci number: 2
4'th fibonacci number: 3
5'th fibonacci number: 5
6'th fibonacci number: 8
7'th fibonacci number: 13
8'th fibonacci number: 21
9'th fibonacci number: 34
```

Time Complexity Analysis

⇒ Let’s draw the recursion tree for this function.



⇒ Each function call, calls two other functions.
First Level: 1 node
Second Level: 2 nodes (1 node has two childs, 1x2 = 2)
Third Level: 4 nodes (both 2 nodes has their 2 childs, 2x2 = 4)
Fourth Level: 8 nodes (all 4 nodes has their 2 childs, 4x2 = 8)
i'th Level: $2^{(i-1)}$

⇒ So, we need to figure out how many levels there will be in the recursion.
Let's assume each time, the argument decreases by one. (Which is not true because one function call decreases by one, and another decreases by two. But, for the sake of simplicity we consider that both function calls decrease the argument by one.)
Therefore, each node in the tree will have two children until the value of n becomes 0 or 1 (base case).
So, to reach from n to 0, with decreasing value one each iteration. We need n steps.
⇒ There will be n levels in the recursion tree.

⇒ **Time Complexity**: $O(2^n)$
Note: the notation used is Big-O and not theta, because of our assumption.

Top-Down

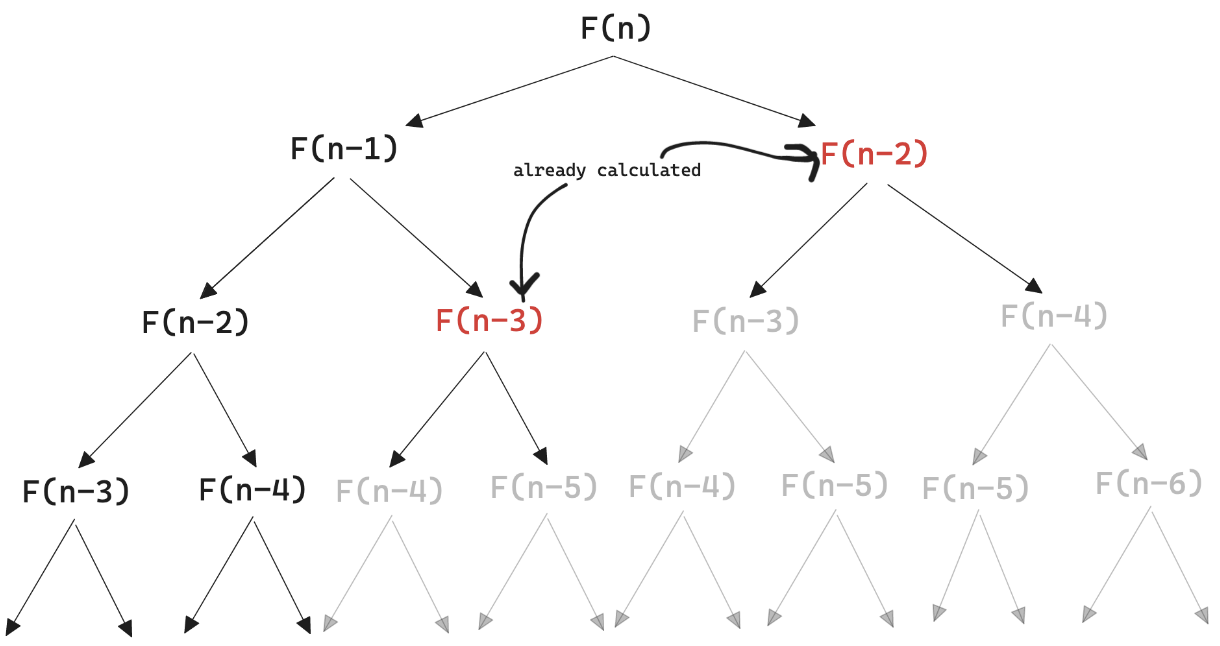
⇒ As you can see in the recursion tree, the function calls are duplicate. (F(n-2) is called multiple times, F(n-3) is called multiple times...)
⇒ So, we have redundant function calls. Because if the argument is the same then it will always give the same result.
⇒ So, instead of calculating the answer for those functions multiple times, we just calculate it one time and then store the value. Later when we call the function we check if we have already calculated the value. If yes, then just use that value, otherwise we need to calculate it.
⇒ This is called "memoization".

⇒ Map data-structure will be the best to use here. As we also need to store the arguments to the functions so that we can later query it using the arguments.
⇒ In this problem, the argument is "which i'th fibonacci number we want".
⇒ So, in the map, we will have values like, {2->1, 3->2, 4->3, 5->5, 6->8} (key is argument, value is the n'th fibonacci number)

```cpp
map<int,int> dp;
int F(int n){
      // base cases
      if(n==0)
      return 0;
      if(n==1)
      return 1;

      // check if already calculated
      if(dp.find(n) != dp.end())
      return dp[n];

      // equation
      int answer = F(n-1) + F(n-2);

      // update the answer if not calculated
      dp[n] = answer;
```

```
        // return answer
        return answer;
}
```

Time Complexity Analysis



⇒ The grayed area won't be running. So, obviously time complexity will be comparatively less than previous.
⇒ Now, because of the early return statement because of memoization, we can't simply figure out the time complexity.

⇒ Every function will only be run once. And there are only n different functions in the recursion tree. F(n), F(n-1), F(n-2), ..., F(0)

⇒ Time Complexity: O(n)

Bottom-Up

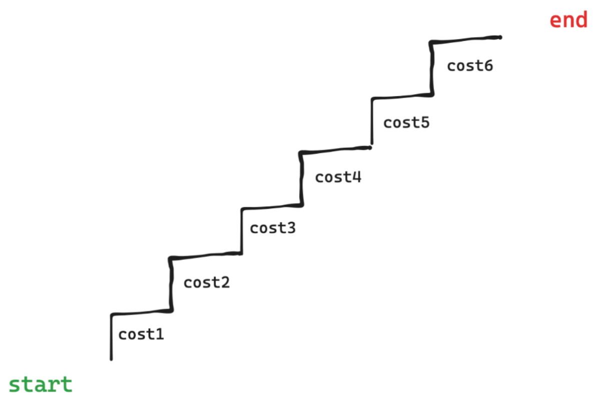
⇒ Recursion uses stack space and generally not having recursion is always better. So, the bottom-up approach removes the need for recursion.

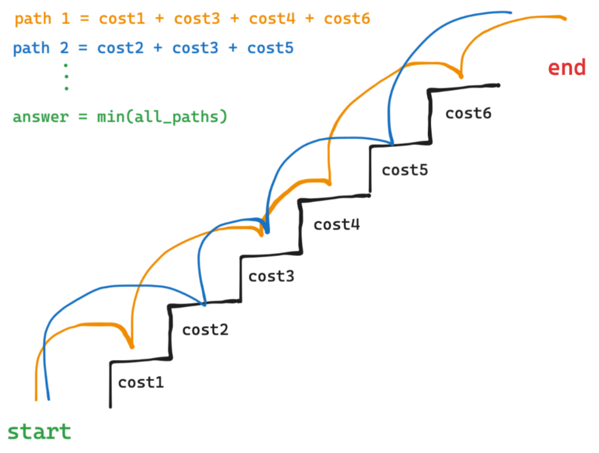
⇒ Now, let's start with why we even have recursion in the first place.
F(n) = F(n-1) + F(n-2)
Answer of n, depends on some other values, in this case, answer of n-1 and n-2. So, we first need to calculate those values.
And recursion can help us here because that's what they do. Before completing the current task, it first completes some other task.
We basically calculate the values as we need them.

⇒ But, instead if we change the order of completing these tasks then we have a little bit of an advantage.
We know that (in this case) the answer of n depends on previous values. So, let's first find out the answer for those and store them in an array. Now, when we try to find the answer for n, we have all the values we need.

```
int solve(int n){
      // we need to store 0...n, total of n+1 values
      vector<int> dp(n+1, 0);

      // base case
      dp[0] = 0;
      dp[1] = 1;

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

      return dp[n];
}
```

Min Cost Climbing Stairs




path 1 = cost1 + cost3 + cost4 + cost6
path 2 = cost2 + cost3 + cost5
⋮
answer = min(all_paths)
end
cost6
cost5
cost4
cost3
cost2
cost1
start

Equation

⇒ Equation tells us how we can build final answer based on smaller subproblems

C_i = cost[i] + min(C_{i-1}, C_{i-2})

⇒ Here, C_i means the minimum cost from step i.
The equation basically means that, if we are already at step i, then we pay the cost at step i. and then we have two options,

- either go to next step (i+1)
- or skip the next step and go to the next step (i+2)

And we find out the minimum cost for both of the options and then we pick the minimum of them.

⇒ Now, when we start we are NOT at step 0. we are on the ground. So, again I have two options, go to step 0, or go to step 1.
⇒ So, answer we want is, min(C_0, C_1)

Recursive

```
int solve(vector<int>& cost, int index){
   // base case
```

```
    if(index >= cost.size())
    return 0;

    int ans = cost[index];
    ans += min(solve(cost, index+1), solve(cost, index+2));
    return ans;
}

int minCostClimbingStairs(vector<int>& cost) {
    return min( solve(cost, 0), solve(cost, 1) );
}
```

⇒ The base case is when we reach the top. The answer of reaching top from top is just 0.

Top-Down

```
map<int, int> memo;
int solve(vector<int>& cost, int index){
        // base case
        if(index >= cost.size())
        return 0;

        // if index was found in memo
        if(memo.find(index) != memo.end()){
        return memo[index];
        }
        else{
        int ans = cost[index];
        ans += min(solve(cost, index+1), solve(cost, index+2));

        memo[index] = ans;

        return ans;
        }
}

int minCostClimbingStairs(vector<int>& cost) {
        // memo.clear();
        return min( solve(cost, 0), solve(cost, 1) );
}
```

Bottom-Up

```
int solve(vector<int>& cost){
      int n = cost.size();
      vector<int> dp(n+1, 0);

      for(int i=n-1; i>=0; i--){
      dp[i] = cost[i] + min(dp[i+1], dp[i+2]);
      }

      return min(dp[0], dp[1]);
}

int minCostClimbingStairs(vector<int>& cost) {
      return solve(cost);
}
```