DYNAMIC PROGRAMMING

Instructor: Dr Tarunpreet Bhatia
Assistant Professor, CSED
TIET

Contents

Dynamic programming (DP)

- It is used, when the solution can be recursively described in terms of solutions to subproblems.
- Algorithm finds solutions to subproblems and stores them in memory for later use.
- DP can solve both optimization and counting problems.
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again.

DP Approaches

- Top down/Memoization: Memoization stores the result of expensive function calls (in arrays or objects) and returns the stored results whenever the same inputs occur again. In this way we can remember any values we have already calculated and access them instead of repeating the same calculation.
- Bottom-up/tabulation: Tabulation is usually accomplished through iteration (a loop). Starting from the smallest subproblem, we store the results in a table (an array), do something with the data until we arrive at the solution.

DP Approaches

Top-down

- Top-down is a recursive problem-solving approach.
- Most of the time, top-down approach is easy to implement because we just add an array or a lookup table to store results during recursion.
- The top-down approach is slower than the bottom-up approach because of the overhead of the recursive calls.
- The top-down approach has also the space overhead of the recursion call stack. If the recursion tree is very deep, we may run out of stack space, which will crash your program.

Bottom-up

- Bottom-up is an iterative approach.
- But in bottom-up approach, we need to define an iterative order to fill thetable and take care of several boundary conditions.
- In other words, the bottom-up approach often has much better constant factors since it has nooverhead for recursive calls.

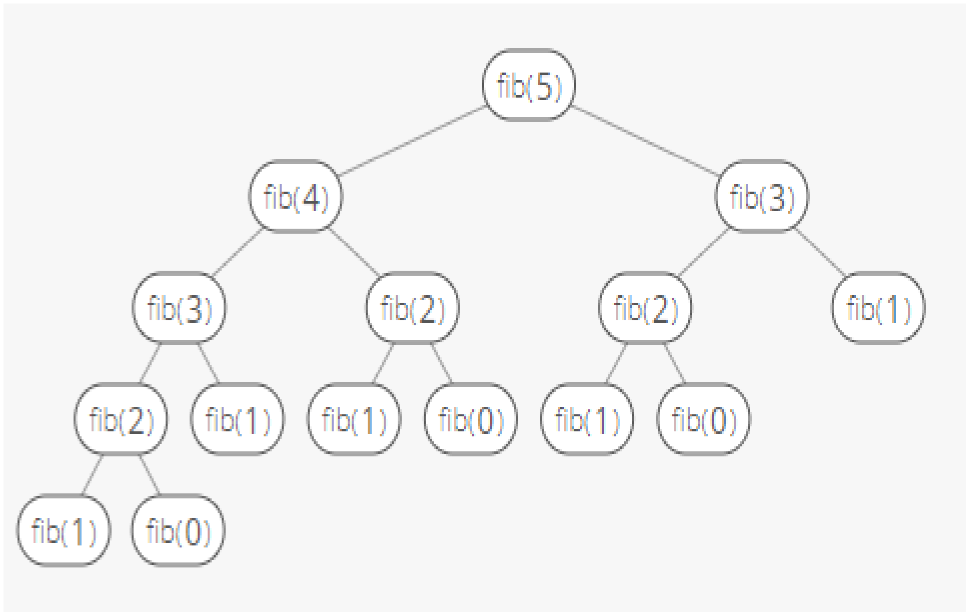
Understanding DP

Fibonacci sequence recursively using the naive approach

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

Recursive Equation

$T(n) = T(n-1) + T(n-2)$



For Worst Case, Let $T(n-1) \approx T(n-2)$

$T(n) = 2T(n-1) + c$

where, $f(n) = O(1)$

$\therefore k=0, a=2, b=1;$

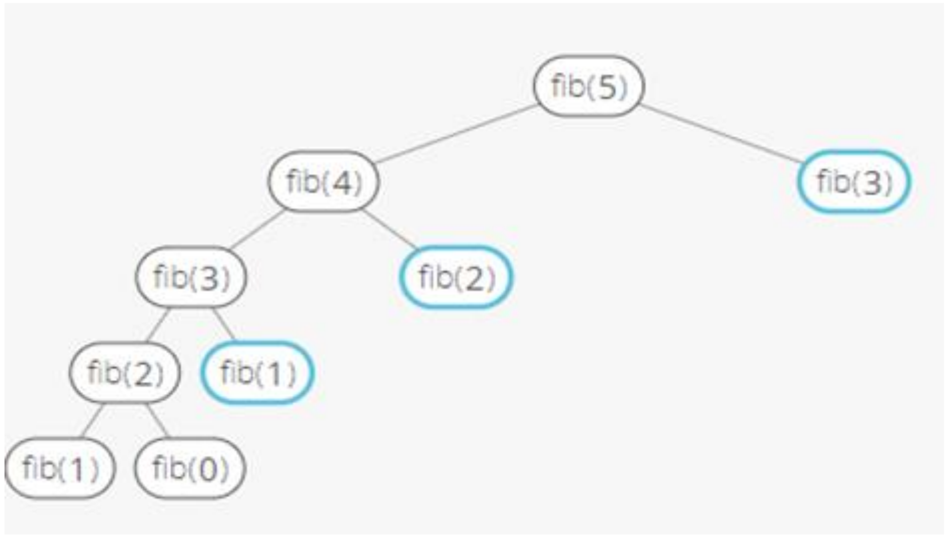
$T(n) = O(n^0 2^{n/1})$

$= O(2^n)$

Understanding DP (contd.)

Top-down fibonacci

```
int dp[n];
int fib(int n)
{
    if(dp[n] != -1)
            return dp[n];
    dp[n] = fib(n-1)+fib(n-2);
    return dp[n];
}
int main(){
for (int i=0;i<=n;i++)
             dp[i] = -1;
dp[0]=0, dp[1]=1;
}
```



Space Complexity: O(n)
Time Complexity: O(n)

Understanding DP (contd.)

Bottom-up fibonacci

```
int dp[n];
int fib(int n)
{
    if (n == 0 || n == 1)
            return n;
    dp[0] = 0;
    dp[1] = 1;
    for (int i=2; i<=n; i++)
         dp[i] = dp[i-1] + dp[i-2]
    return dp[n];
}
```

Space Complexity: O(n)
Time Complexity: O(n)

Understanding DP (contd.)

- The improvement can be reduced to an exponential time solution to a polynomial time solution, with an overhead of using additional memory for storing intermediate results.
- While both have the linear time complexity for Fibonacci sequence, since memoization uses recursion, if you try to find the Fibonacci number for a large n, you will get a stack overflow (maximum call stack size exceeded).
- Tabulation, on the other hand, doesn’t take as much space, and will not cause a stack overflow.

Idea behind DP

- Divide the problem into multiple subproblems and save the result of each subproblem.
- If the same subproblem occurs, rather than calculating it again, we can use the old reference from the previously calculated subproblem.
- Once we have calculated the result for all the subproblems, conquer the result for final output.

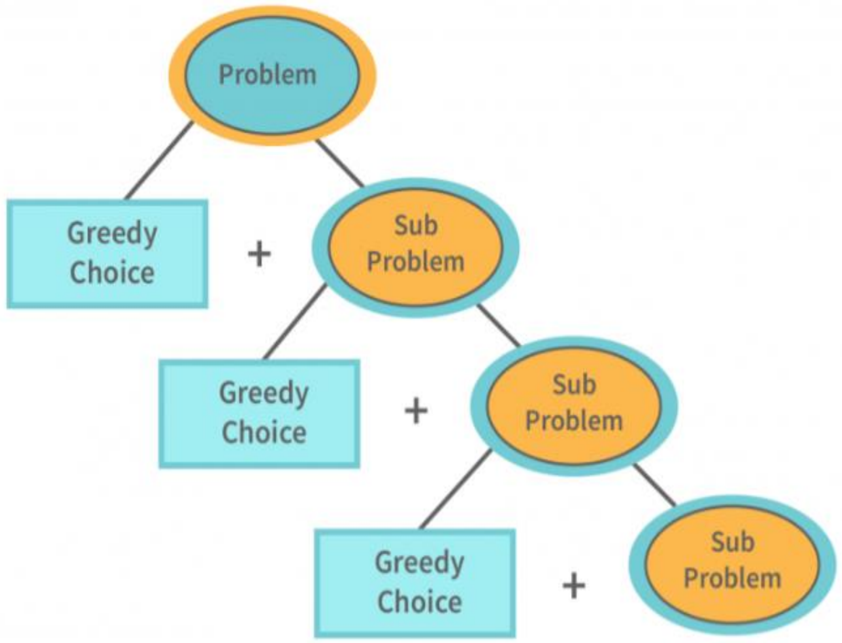
DP and Recursion

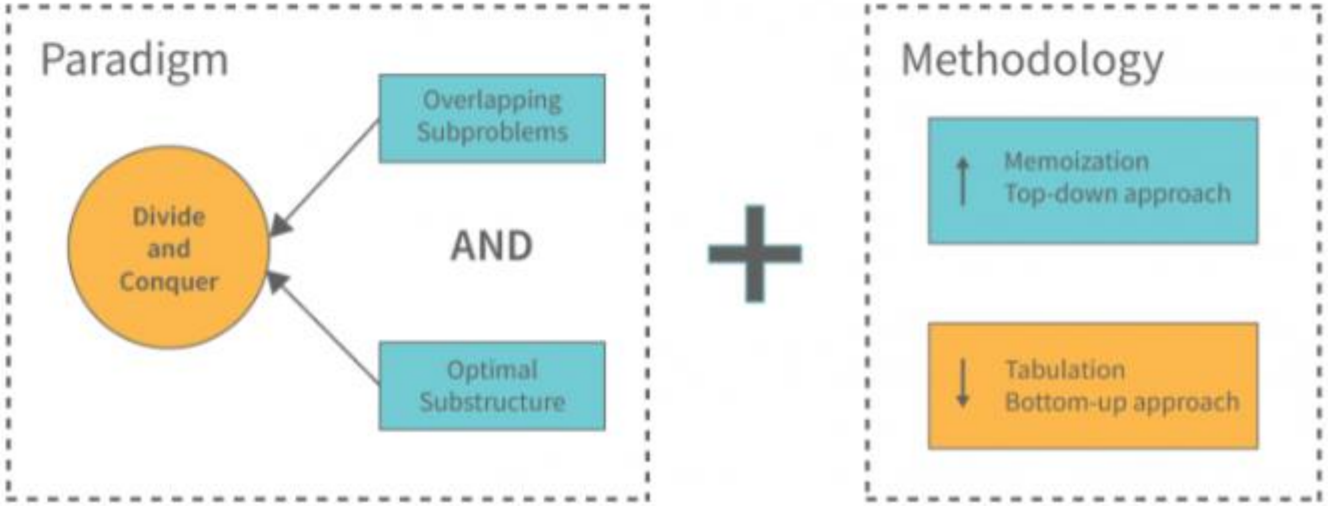
- Dynamic programming is an optimization method used for problem-solving; the problems are broken into smaller subproblems to be solved.
- While recursion is when a function can be called and executed by itself.
- Recursion risks to solve identical subproblems multiple times. This inefficiency is addressed and remedied by dynamic programming.
- Both methods have different uses, but recursion can be used in dynamic programming, even when this can perfectly function without this.

DP and DAC

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Differences between divide-and-conquer (DAC) and DP:
  - DAC: Independent sub-problems, solve sub-problems independently and recursively, (so same sub(sub)problems solved repeatedly)
  - DP: Sub-problems are dependent, i.e., sub-problems share sub-sub-problems, every sub(sub)problem solved just once, solutions to sub(sub)problems are stored in a table and used for solving higher level sub-problems.

DP and Greedy





DP and Greedy

Greedy	DP
We make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	We make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.
There is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
It is more efficient in terms of memory as it never look back or revise previous choices	It requires table for Memoization and it increases it's memory complexity.
Greedy methods are generally faster.	Generally slower
Only considers the current choice without looking about future.	It considers the future choices while selects the current choice.

Properties of a problem that can be solved with dynamic programming

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

How to Learn and Master DP?

- Get a good grip on solving recursive problems.
- Theory of dividing a problem into subproblems is essential to understand.
- Learn to store the intermediate results in the array.
- Solve as many problems as you can. It will give you a significant understanding and logic building for dynamic problems.

DP Solution: A Sequence of 4 Steps

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information