CSE 221: Algorithms

Dynamic Programming

Mumit Khan

Computer Science and Engineering
BRAC University

References

1. Jon Kleinberg and Éva Tardos, *Algorithm Design*. Pearson Education, 2006.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

Last modified: December 30, 2010

Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don't solve the same subproblem twice, but rather save the solution so it can be re-used later on.
- Often used for a large class to optimization problems.

  Unlike Greedy algorithms, implicitly solve all subproblems.
- Divide and conquer algorithms partition the problem into disjoint subproblems.
- Motivating the case for DP with Memoization – a top-down technique, and then moving on to Dynamic Programming – a bottom-up technique.

Recursive solution to Fibonacci numbers

Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

$$\langle 0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89, \ldots \rangle$$

and described by the following recurrence.

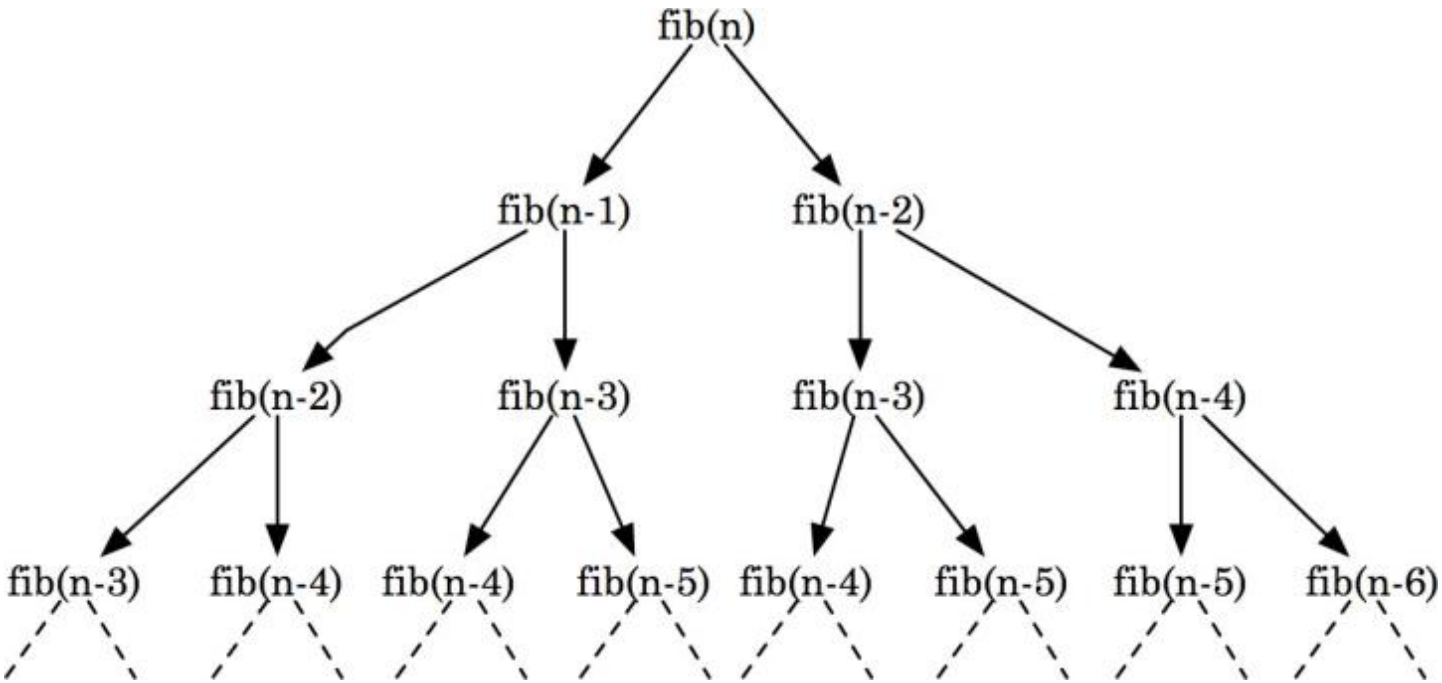
$$\text{FIB}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2) & \text{if } n \geq 2 \end{cases}$$

Straightforward recursive algorithm

FIBONACCI(n) $\quad D\ n \geq 0$

```
1  if n = 0 or n = 1
2      then return n
3      else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Recursion tree

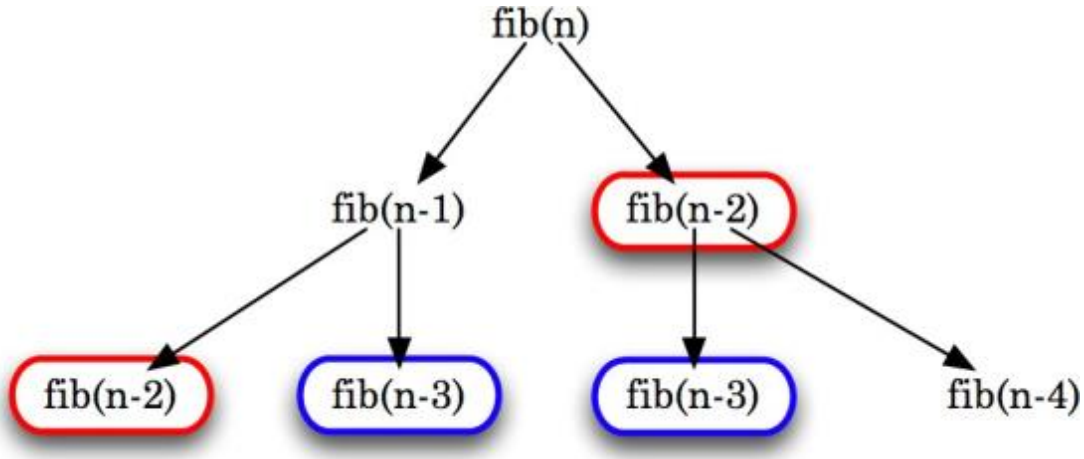


Complexity

This recursive algorithm for Fibonacci numbers has exponential running time!

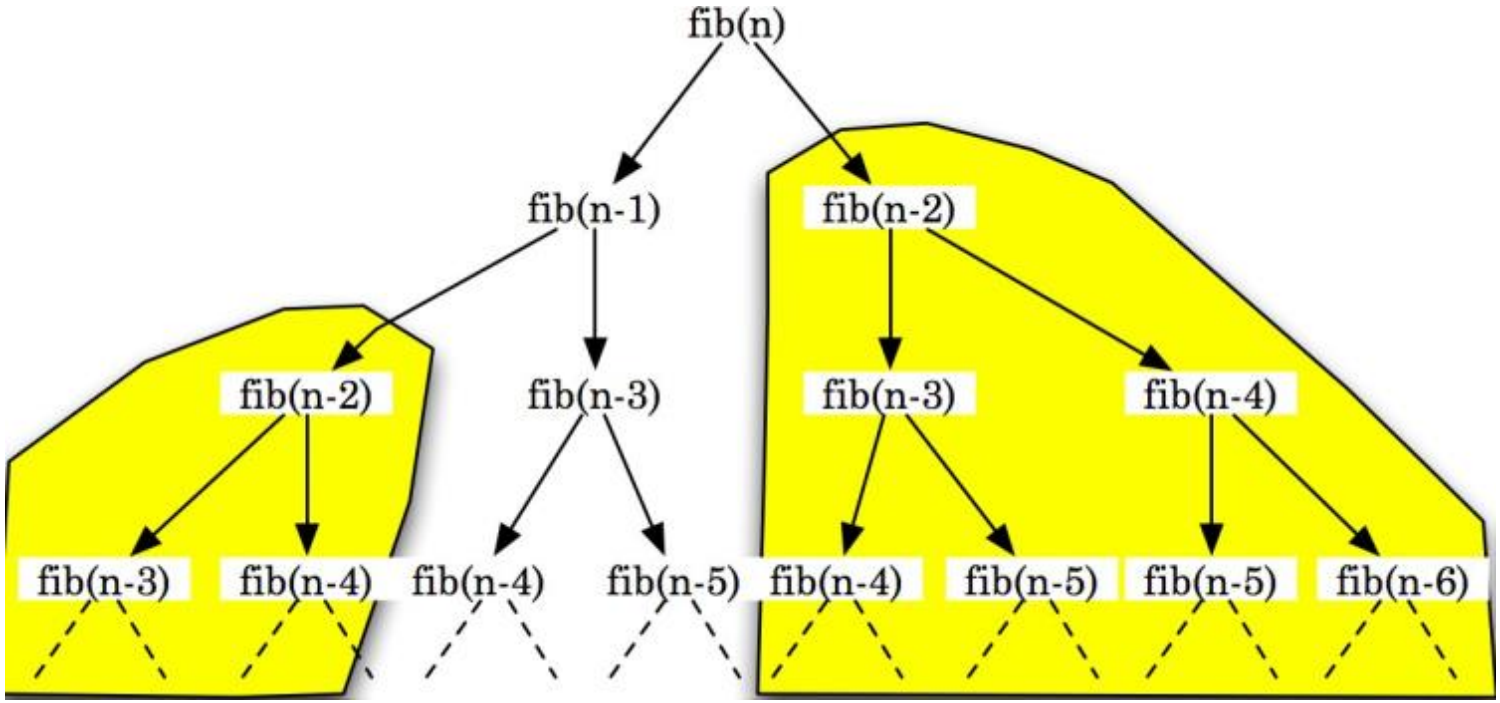
To be precise, $T(n) = O(\phi^n)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Redundant computations



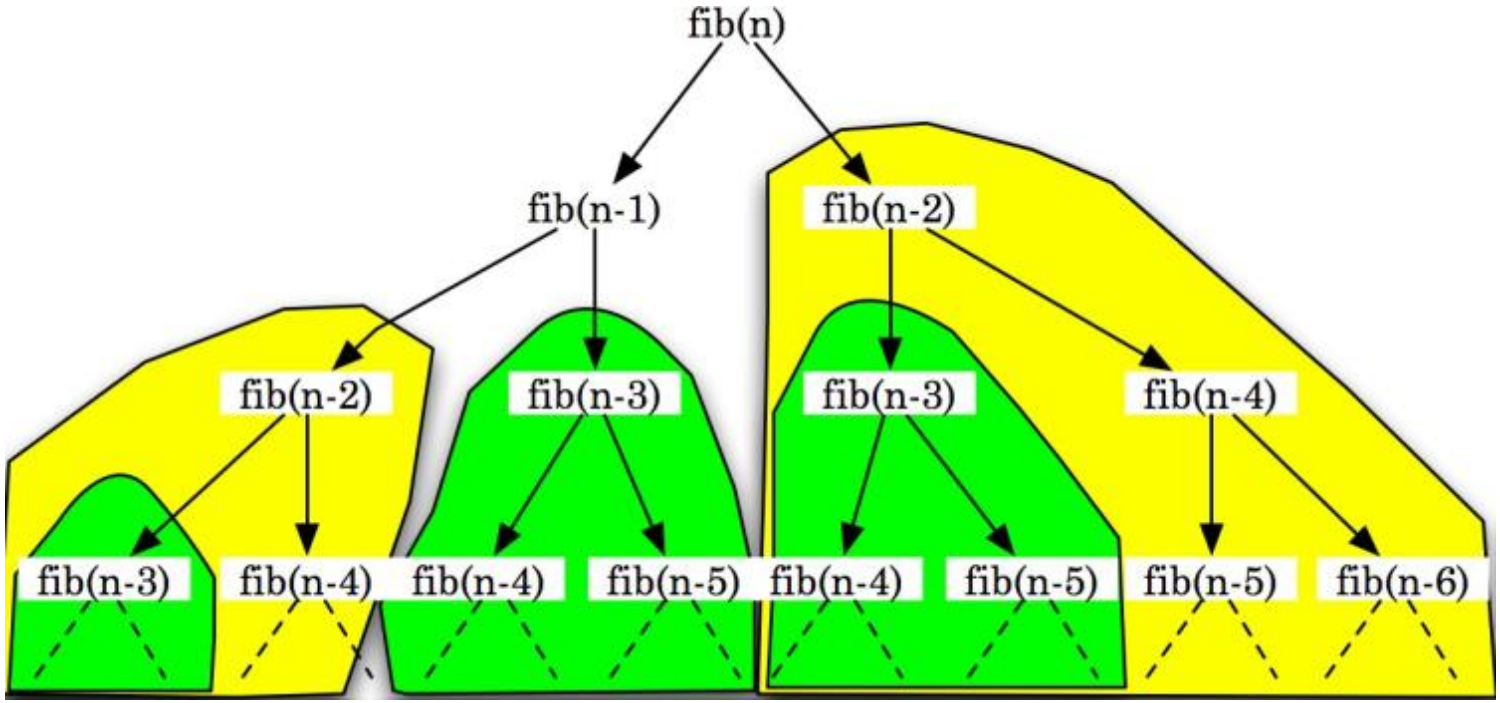
Note how $\text{FIB}(n-2)$ and $\text{FIB}(n-3)$ are each being computed twice.

Redundant computations



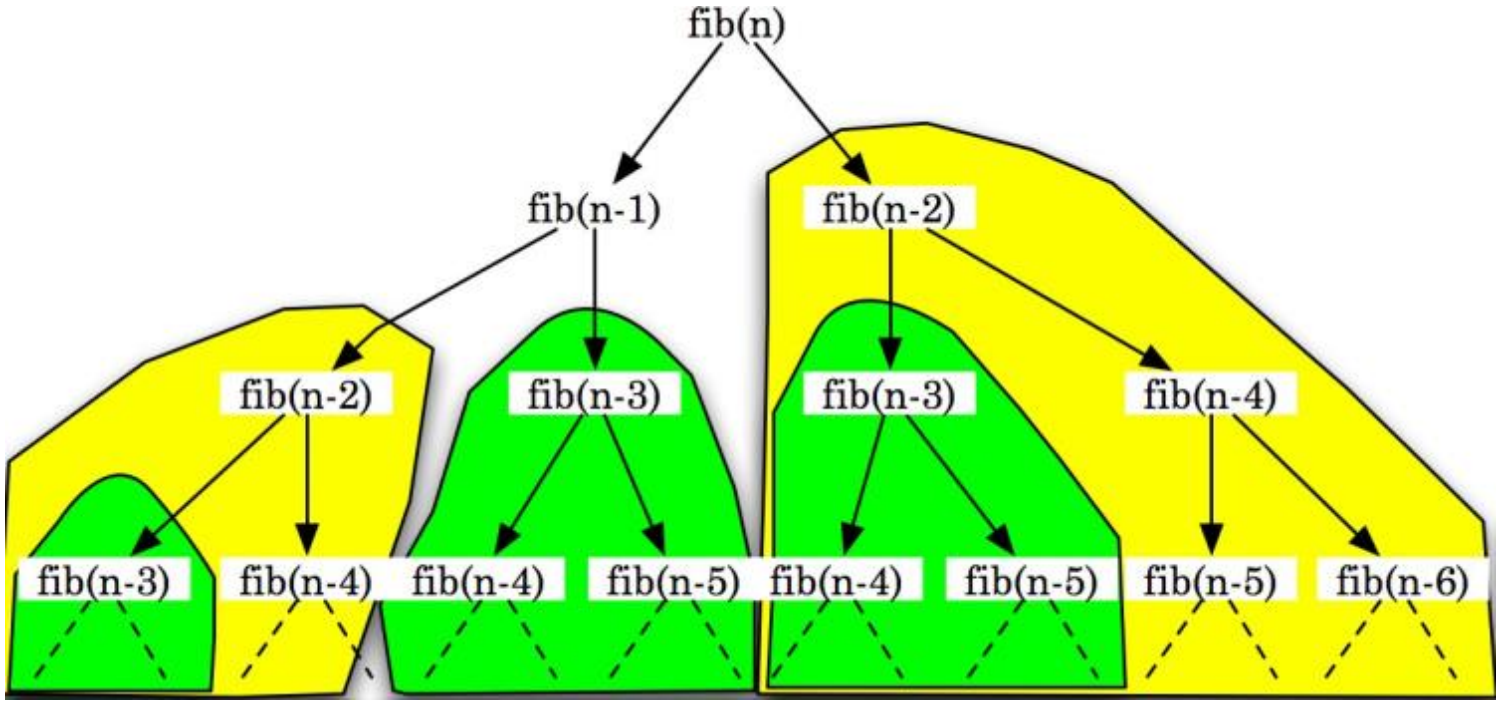
In fact, computing $\text{FIB}(n-2)$ involves computing a whole subtree.

Redundant computations



Likewise for computing $\text{FIB}(n-3)$.

Redundant computations



Observations

- Spectacular redundancy in computation – how many times are we computing $\text{FIB}(n-2)$? $\text{FIB}(n-3)$?
- What if we compute and save the result of $\text{FIB}(i)$ for $i = \{2, 3, \ldots, n\}$ the first time, and then re-use it each time afterward?
- Ah, we've just (re)discovered Memo(r)ization!

Memoization

Definition (Memoization)

The process of saving solutions to subproblems that can be re-used later without redundant computations.

Basic idea

Typically, the solutions to subproblems (i.e., the intermediate solutions) are saved in a global array, which are later looked up and re-used as needed.

1. At each step of computation, first see if the solution to the subproblem has already been found and saved.
2. If so, simply return the solution.
3. If not, compute the solution, and save it before returning the solution.

Memoized recursive algorithm for Fibonacci numbers

M-FIBONACCI(n) $\quad$ D $n \geq 0$, global $F = [0..n]$

```
1  if n = 0 or n = 1
2      then return n                          D Our base conditions.
3  if F[n] is empty                           D No saved solution found for n.
4      then F[n] ← m-fIBONACCI(n − 1) + m-fIBONACCI(n − 2)
5  return F[n]
```

Questions

- What is this global array F? It's used store the values of the intermediate results, and must be initialized by the caller to all empty.
- What is an appropriate sentinel to indicate that $F[i], 0 \leq i \leq n$ has not been solved yet (i.e., empty)? Use -1, which is guaranteed to be an invalid value.

Memoized ...Fibonacci numbers (continued)

```
FIBONACCI(n)                    ¢ n ≥ 0
    ¢ Allocate an array F[0..n] to save results (length[F] = n+1).
1   for i ← 0 to n
2       do F[i] ← −1            ¢ No solution computed for i yet (sentinel)
3   return m-FIBONACCI(F, n)
```

```
M-FIBONACCI(F, n)               ¢ n ≥ 0, F = [0..n]
1   if n ≤ 1
2       then return n
3   if F[n] = −1                ¢ No saved solution found for n.
4       then F[n] ← m-FIBONACCI(F, n − 1) + m-FIBONACCI(F, n − 2)
5   return F[n]
```

Running time

Each element $F[2] \ldots F[n]$ is filled in just once in $\Theta(1)$ time, so

$T(n) = \Theta(n)$.

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).
- Any recursive algorithm where redundant solutions are computed, Memoization is an appropriate solution.

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).
- Any recursive algorithm where redundant solutions are computed, Memoization is an appropriate solution.
- Often called Top-down Dynamic Programming.

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).
- Any recursive algorithm where redundant solutions are computed, Memoization is an appropriate solution.
- Often called Top-down Dynamic Programming.

Questions to ask (and remember)

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).
- Any recursive algorithm where redundant solutions are computed, Memoization is an appropriate solution.
- Often called Top-down Dynamic Programming.

Questions to ask (and remember)

- What are the drawbacks, if any, of memoization?

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).
- Any recursive algorithm where redundant solutions are computed, Memoization is an appropriate solution.
- Often called Top-down Dynamic Programming.

Questions to ask (and remember)

- What are the drawbacks, if any, of memoization?
- Would all recursive algorithms benefit from memoization?

Memoization highlights

- Idea is to re-use saved solutions, trading off space for time.
- Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are overlapping subproblems).
- Any recursive algorithm where redundant solutions are computed, Memoization is an appropriate solution.
- Often called Top-down Dynamic Programming.

Questions to ask (and remember)

- What are the drawbacks, if any, of memoization?
- Would all recursive algorithms benefit from memoization? For example, would the recursive algorithm to compute the factorial of a number benefit from memoization?

Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number *n* top down by computing (and saving) solutions for smaller values.

Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number n top down by computing (and saving) solutions for smaller values.
- Idea: why not build up the solution bottom-up, starting from the base case(s) all the way to n?

Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number n top down by computing (and saving) solutions for smaller values.
- Idea: why not build up the solution bottom-up, starting from the base case(s) all the way to n?
- This bottom up construction gives us the first Dynamic Programming algorithm.

Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number n top down by computing (and saving) solutions for smaller values.
- Idea: why not build up the solution bottom-up, starting from the base case(s) all the way to n?
- This bottom up construction gives us the first Dynamic Programming algorithm.

Dynamic programming algorithm for fibonacci numbers

FIBONACCI(n) $\quad D\ n \geq 0$

```
1  F[0] ← 0
2  F[1] ← 1
3  for i ← 2 to n
4        do F[i] ← F[i − 1] + F[i − 2]
5  return F[n]
```

Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number n top down by computing (and saving) solutions for smaller values.
- Idea: why not build up the solution bottom-up, starting from the base case(s) all the way to n?
- This bottom up construction gives us the first Dynamic Programming algorithm.

Dynamic programming algorithm for fibonacci numbers

FIBONACCI(n) $\quad D\ n \geq 0$

```
1  F[0] ← 0
2  F[1] ← 1
3  for i ← 2 to n
4        do F[i] ← F[i − 1] + F[i − 2]
5  return F[n]
```

$T(n) = \Theta(n)$

Dynamic programming (continued)

The pattern

1. **Formulate the problem recursively**. Write a formula for the whole problem as a simple combination of of the answers to smaller subproblems.
2. **Build solutions to the recurrence from the bottom up**. Write an algorithm that starts with the base case, and works its way up to the final solution by considering the subproblems in the correct order.

Observations

1. Must ensure that the recurrence is correct of course!
2. Need a "place" to store the solutions to subproblems, and need to look these solutions up when needed. Typically, but not always, a multi-dimensional table is used as storage.

Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.
- DP is not applicable to all optimization problems.
- Two key ingredients optimization problems must have in order for dynamic programming to apply:

Optimal Substructure: A problem has optimal substructure if an optimal solution can be constructed efficiently from optimal solution of its subproblems.

Overlapping Subproblems: A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times.

Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.
- DP is not applicable to all optimization problems.
- If a problem has the following properties, then it's likely to have a dynamic programming solution.

  Polynomially many subproblems The total number of subproblems should be a polynomial, or else DP may not provide an efficient solution.

  Subproblem optimality If the optimal solution to the entire problem contain optimal solution to the subproblems, then it has the subproblem optimality property. Also called the *principle of optimality*.

Dynamic Programming highlights

- Dynamic Programming, just like Memoization, avoids computing solutions to overlapping subproblems by saving intermediate results, and thus both require space for the "table".
- Dynamic Programming is a bottom-up techniques, and finds the solution by starting from the base case(s) and works its way upwards.
- Developing a Dynamic Programming solution often requires some thought into the subproblems, especially how to find the natural order in which to solve the subproblems.
- Unlike Memoization, which solves only the needed subproblems, DP solves all the subproblems, because it does it bottom-up.
- Dynamic Programming on the other hand may be much more efficient because its iterative, whereas Memoization must pay for the (often significant) overhead due to recursion.

Conclusion

- Memoization is the top-down technique, and dynamic programming is a bottom-up technique.
- The key to Dynamic programming is in “intelligent” recursion (the hard part), not in filling up the table (the easy part).
- Dynamic Programming has the potential to transform exponential-time brute-force solutions into polynomial-time algorithms.
- Greed does not pay, Dynamic Programming does!