Dynamic Programming

Dynamic Programming (1)

Oct 4th, 2018

Algorithm Design and Analysis

YUN-NUNG (VIVIAN) CHEN HTTP://ADA.MIULAB.TW

國立臺灣大學
National Taiwan University

Slides credited from Hsueh-I Lu, Hsu-Chun Hsiao, & Michael Tsai

Outline

- Dynamic Programming
- DP #1: Rod Cutting
- DP #2: Stamp Problem
- DP #3: Matrix-Chain Multiplication
- DP #4: Sequence Alignment Problem
  - Longest Common Subsequence (LCS) / Edit Distance
  - Viterbi Algorithm
  - Space Efficient Algorithm
- DP #5: Weighted Interval Scheduling
- DP #6: Knapsack Problem
  - 0/1 Knapsack
  - Unbounded Knapsack
  - Multidimensional Knapsack
  - Fractional Knapsack

Algorithm Design Strategy

- Do not focus on "specific algorithms"
- But "some strategies" to "design" algorithms

- First Skill: Divide-and-Conquer (各個擊破)
- Second Skill: Dynamic Programming (動態規劃)

4

Dynamic Programming

Textbook Chapter 15 – Dynamic Programming

Textbook Chapter 15.3 – Elements of dynamic programming

What is Dynamic Programming?

- Dynamic programming, like the divide-and-conquer method, solves problems by <u>combining the solutions to subproblems</u>
  - 用空間換取時間
  - 讓走過的留下痕跡
- “Dynamic”: time-varying
- “Programming”: a *tabular* method

Dynamic Programming: planning over time

Algorithm Design Paradigms

- Divide-and-Conquer
  - partition the problem into **independent** or **disjoint** subproblems
  - repeatedly solving the common subsubproblems

  → more work than necessary

- Dynamic Programming
  - partition the problem into **dependent** or **overlapping** subproblems
  - avoid recomputation
    - ✓ Top-down with memoization
    - ✓ Bottom-up method

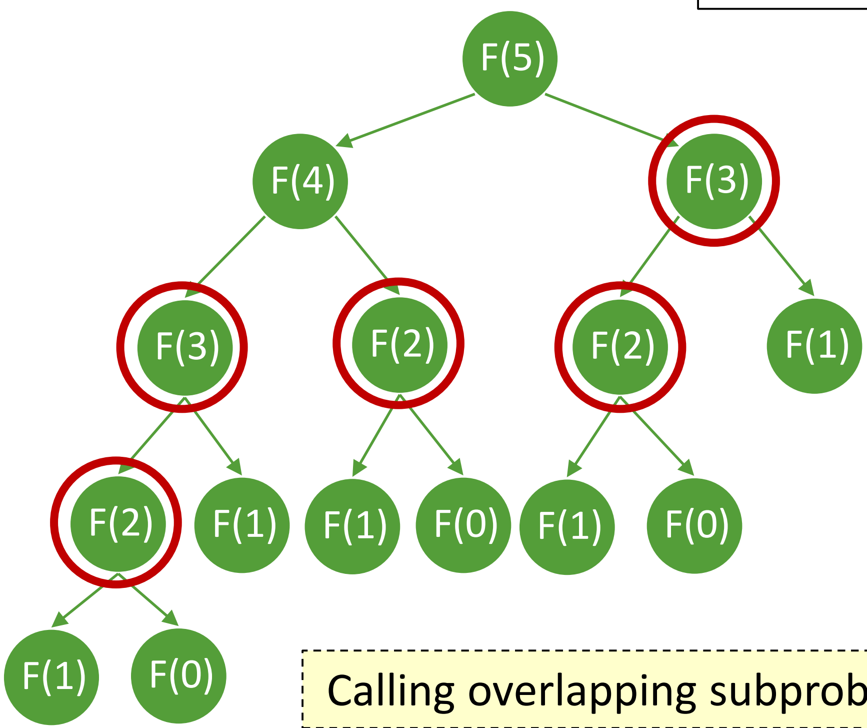
Dynamic Programming Procedure

- Apply four 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, typically in a **bottom-up** fashion
  4. Construct an optimal solution from computed information

Rethink Fibonacci Sequence

- Fibonacci sequence (費波那契數列)
  - Base case: F(0) = F(1) = 1
  - Recursive case: F(n) = F(n-1) + F(n-2)

```
Fibonacci(n)
  if n < 2 // base case
    return 1
  // recursive case
  return Fibonacci(n-1)+Fibonacci(n-2)
```



✓ F(3) was computed twice

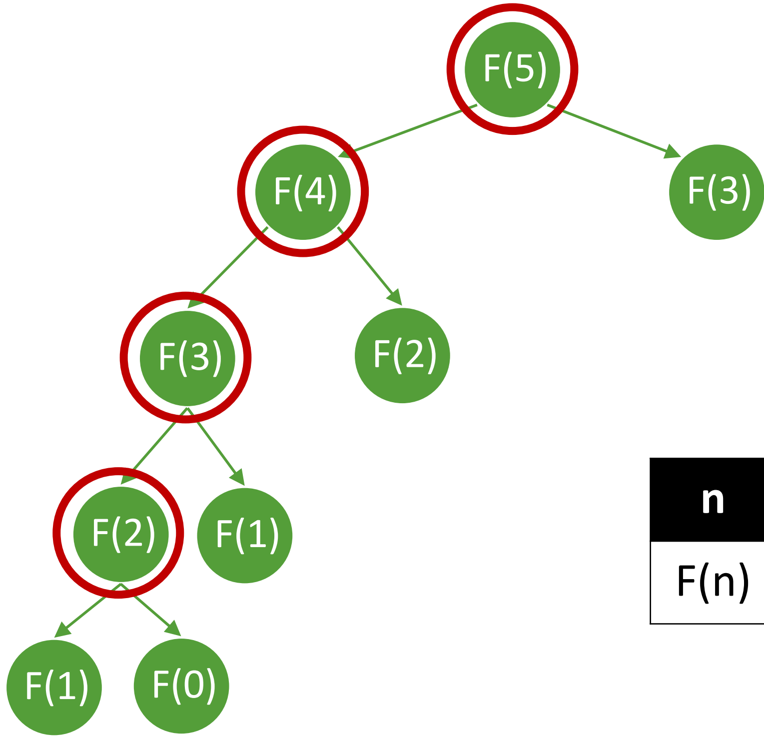
✓ F(2) was computed 3 times

$$T(n) = O(2^n)$$

Calling overlapping subproblems result in poor efficiency

Fibonacci Sequence
Top-Down with Memoization

- Solve the overlapping subproblems recursively with memoization
  - Check the memo before making the calls



n	0	1	2	3	4	5
F(n)	1	1	2	3	5	8

Avoid recomputation of the same subproblems using memo

Fibonacci Sequence

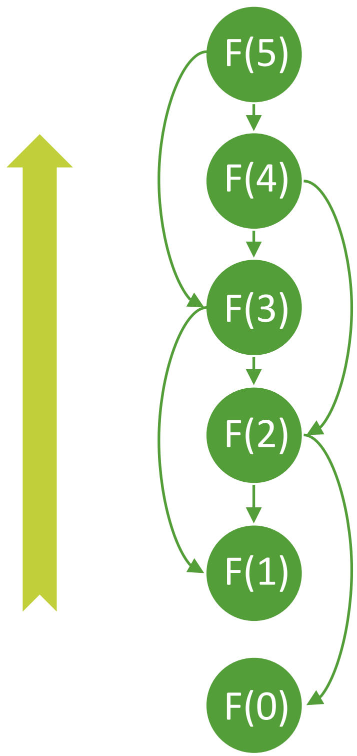
Top-Down with Memoization

```
Memoized-Fibonacci(n)
  // initialize memo (array a[])
  a[0] = 1
  a[1] = 1
  for i = 2 to n
    a[i] = 0
  return Memoized-Fibonacci-Aux(n, a)

Memoized-Fibonacci-Aux(n, a)
  if a[n] > 0
    return a[n]
  // save the result to avoid recomputation
  a[n] = Memoized-Fibonacci-Aux(n-1, a) + Memoized-Fibonacci-Aux(n-2, a)
  return a[n]
```

Fibonacci Sequence
Bottom-Up Method

- Building up solutions to larger and larger subproblems




```
Bottom-Up-Fibonacci(n)
  if n < 2
    return 1
  a[0] = 1
  a[1] = 1
  for i = 2 … n
    a[i] = a[i-1] + a[i-2]
  return a[n]
```

Avoid recomputation of the same subproblems

Optimization Problem

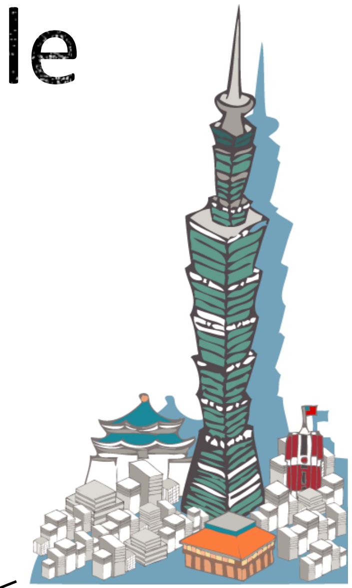
- Principle of Optimality
  - Any subpolicy of an optimum policy must itself be an optimum policy with regard to the initial and terminal states of the subpolicy
- Two key properties of DP for optimization
  - Overlapping subproblems
  - Optimal substructure – an optimal solution can be constructed from optimal solutions to subproblems
    - ✓ Reduce search space (ignore non-optimal solutions)

If the optimal substructure (principle of optimality) does not hold, then it is incorrect to use DP

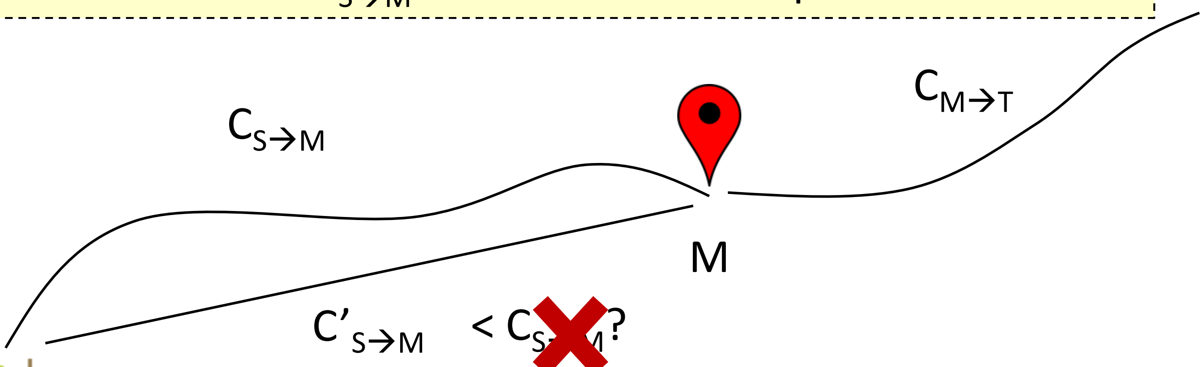
Optimal Substructure Example

- Shortest Path Problem
  - Input: a graph where the edges have positive costs
  - Output: a path from S to T with the smallest cost

The path costing $C_{S\rightarrow M} + C_{M\rightarrow T}$ is the shortest path from S to T
$\rightarrow$ The path with the cost $C_{S\rightarrow M}$ must be a shortest path from S to M



Taipei (T)



Tainan (S)

Proof by "Cut-and-Paste" argument (proof by contradiction):
Suppose that it exists a path with smaller cost $C'_{S\rightarrow M}$, then we can "cut" $C_{S\rightarrow M}$ and "paste" $C'_{S\rightarrow M}$ to make the original cost smaller

14 To Be Continued…

Question?

Important announcement will be sent to @ntu.edu.tw mailbox
& post to the course website

Course Website: http://ada.miulab.tw

Email: ada-ta@csie.ntu.edu.tw