

Dynamic programming

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

-Dynamic Programming

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Dynamic Programming

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.
- *Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-instances.*
- *Dynamic programming is one of the algorithmic paradigm that solves many problems that are failed under other paradigms such as Divide and Conquer, Greedy approach etc.*
- “Programming” here means “**planning**”
- **Main idea:**
 - ☐ set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
 - ☐ solve smaller instances once
 - ☐ record solutions in a table
 - ☐ extract solution to the initial instance from that table

Introduction

- Dynamic Programming(DP) applies to optimization problems in which a set of choices must be made in order to arrive at an optimal solution.
- As choices are made, subproblems of the same form arise.
- DP is effective when a given problem may arise from more than one partial set of choices.
- The *key technique* is to store the solution to each subproblem in case it should appear
- Divide and Conquer algorithms partition the problem into independent subproblems.
- Dynamic Programming is applicable when the subproblems are not independent.(In this case DP algorithm does more work than necessary)
- Dynamic Programming algorithm solves every subproblem just once and then saves its answer in a table.

Four steps for the development of a dynamic-programming algorithm

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. Compute an optimal solution from computed/stored information.
- Steps 1-3 form the basis of a dynamic-programming solution to a problem.
 - Step 4 can be omitted if only the value of an optimal solution is required.
 - When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

Dynamic Programming

- It uses a “bottom-up” approach in that the subproblems are arranged and solved in a systematic fashion, which leads to a solution to the original problem.
- This bottom-up approach implementation is more efficient than a “top-down” counterpart mainly because duplicated computation of the same problems is eliminated.
- This technique is typically applied to solving optimization problems, although it is not limited to only optimization problems.

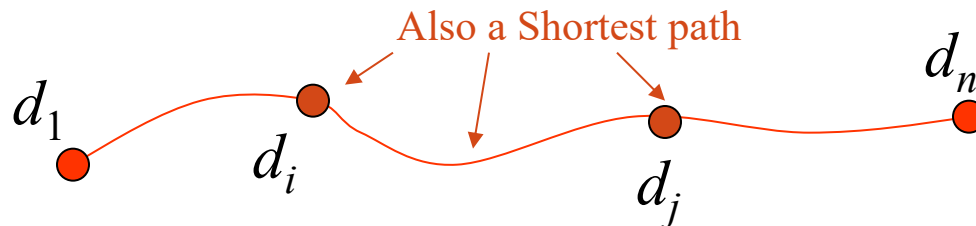
Dynamic programming typically involves two steps:

- (1) **develop a recursive strategy for solving the problem**
- (2) **develop a “bottom-up” implementation without recursion.**

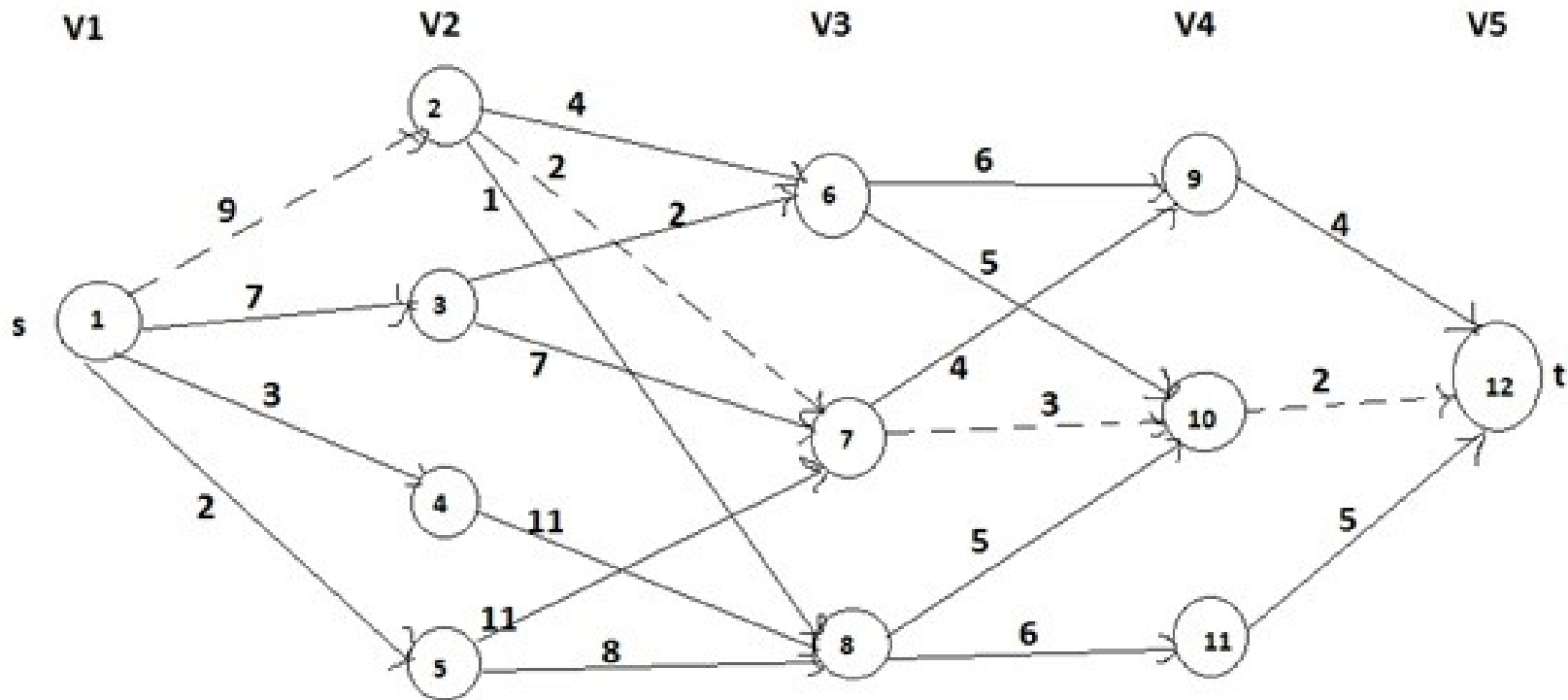
The Principle of Optimality:

In solving optimization problems which require making a sequence of decisions, such as the change making problem, we often apply the following principle in setting up a recursive algorithm: Suppose an optimal solution made decisions d_1, d_2 , and ..., d_n . The subproblem starting after decision point d_i and ending at decision point d_j , also has been solved with an optimal solution made up of the decisions d_i through d_j . That is, any subsequence of an optimal solution constitutes an optimal sequence of decisions for the corresponding **subproblem**. This is known as the principle of optimality which can be illustrated by the shortest paths in weighted graphs as follows:

A shortest path from d_1 to d_n



Multistage Graph



MULTI STAGE GRAPH

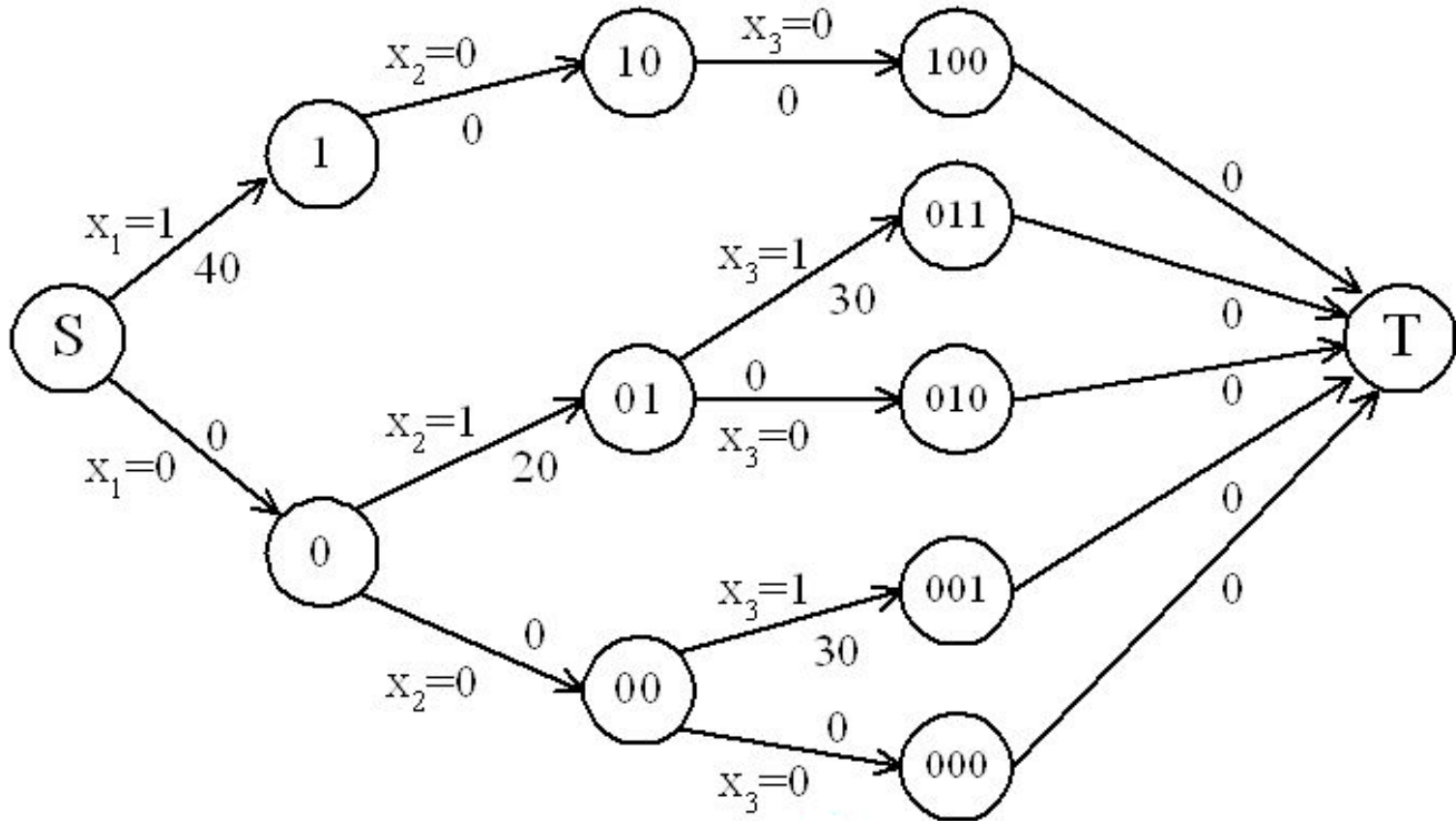
A **Multistage graph** is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).

The 0–1 Knapsack Problem:

Given n objects 1 through n , each object i has an integer weight w_i and a real number value v_i , for $1 \leq i \leq n$. There is a knapsack with a total integer capacity W . The 0–1 knapsack problem attempts to fill the sack with these objects within the weight capacity W while **maximizing** the total value of the objects included in the sack, where an object is totally included in the sack or no portion of it is in at all. That is, solve the following optimization problem with $x_i = 0$ or 1 , for $1 \leq i \leq n$:

$$\text{Maximize } \sum_{i=1}^n x_i v_i \text{ subject to } \sum_{i=1}^n x_i w_i \leq W.$$

0/1 Knapsack Problem by Multistage Graph

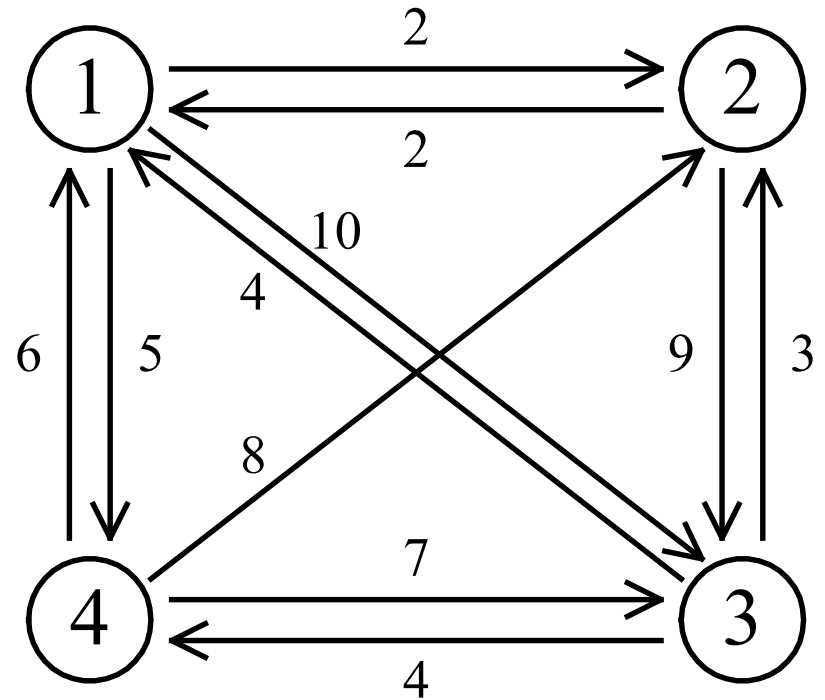


i	W_i	P_i
1	10	40
2	3	20
3	5	30

W=10

Traveling salesperson problem

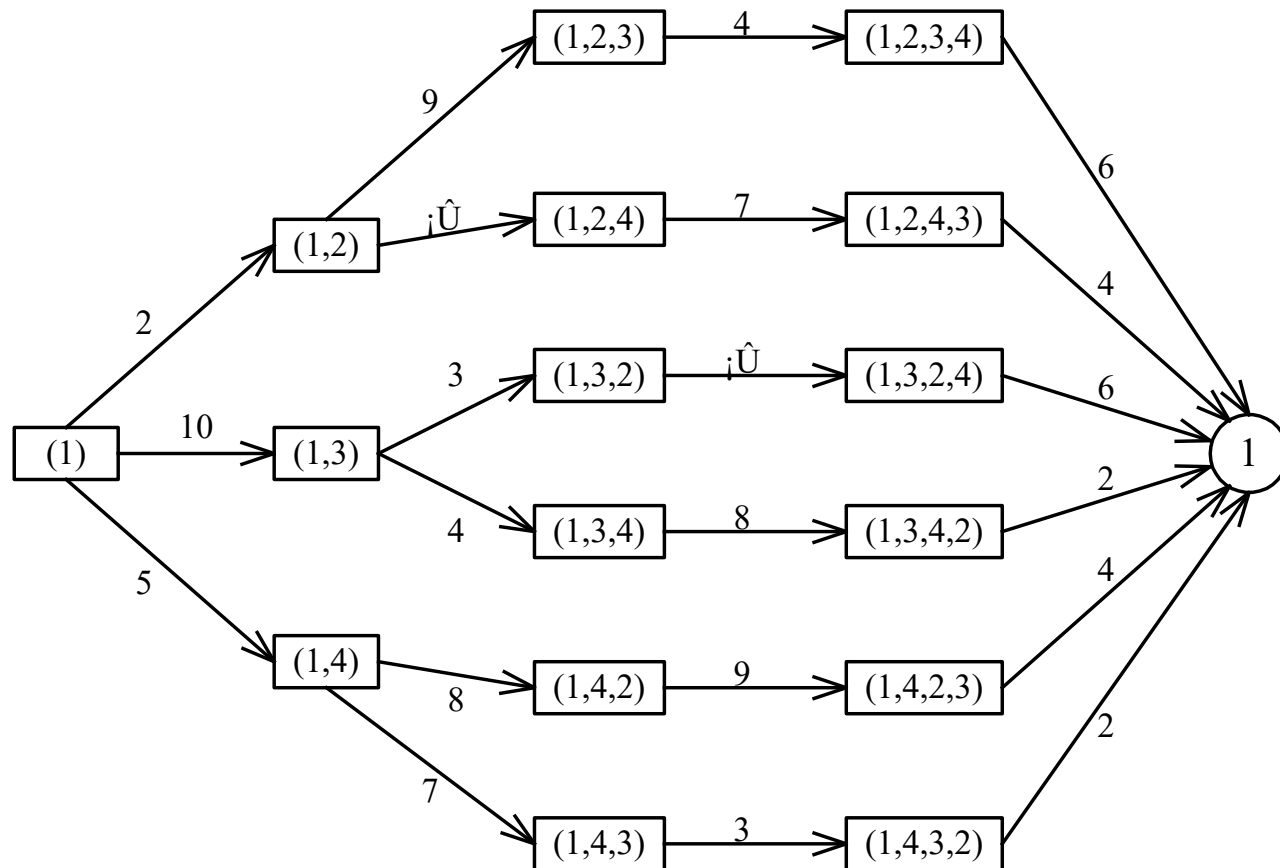
- e.g. a directed graph :



- Cost matrix:

	1	2	3	4
1	∞	2	10	5
2	2	∞	9	∞
3	4	3	∞	4
4	6	8	7	∞

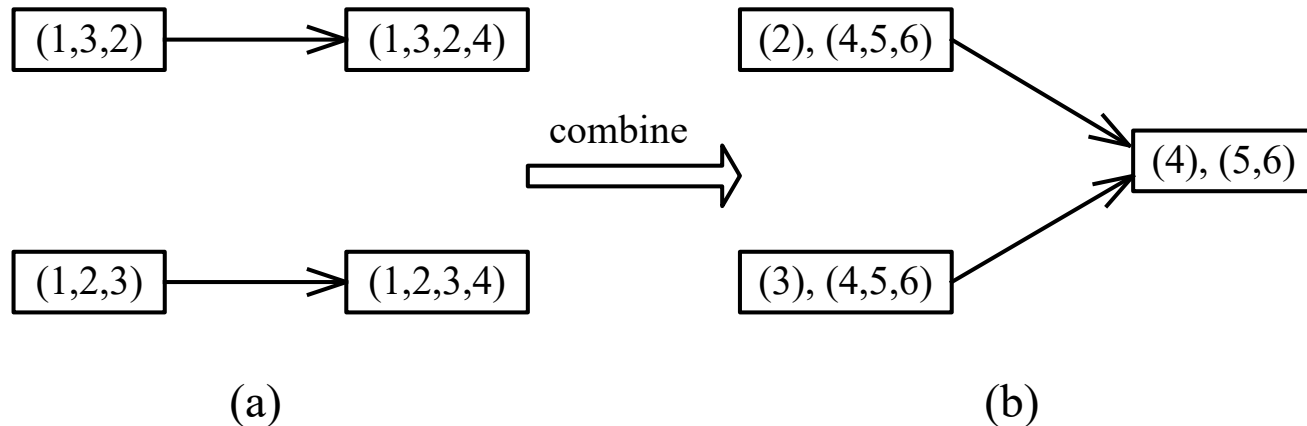
The multistage graph solution



- A multistage graph can describe all possible tours of a directed graph. Find the shortest path: (1, 4, 3, 2, 1)
 $5+7+3+2=17$

Representation of a node

- Suppose that we have 6 vertices in the graph.
- We can combine $\{1, 2, 3, 4\}$ and $\{1, 3, 2, 4\}$ into one node.



- $(3), (4,5,6)$ means that the last vertex visited is 3 and the remaining vertices to be visited are (4, 5, 6).

Approaches to solve a Dynamic Programming problems:

1. Top down DP

- In this approach, a recursive call is made with the given problem size and next call is made with the smaller subproblem (argument value gets decreased) and further calls will be smaller than the previous ones just like calls in the Fibonacci series.
- Once smallest subproblem is computed, then, its result will be used for the next calling. It's comparatively easy to think and codes look simple, but, memory consumption is high and written codes are difficult to understand for a reader. This is referred as **Memoization Method**.

2. Bottom up DP

- In this approach, rather than beginning with the main problem, start solving with the smallest one and on the way increase the sub-problem size through using the computed sub problems. It's based on iteration rather than recursion. It's difficult to think, but, it's easy for a reader to understand and also, memory consumption is less. Therefore, it's more suggested to use. This is referred as **Tabulization Method**.

Comparing Approaches to solve a Dynamic

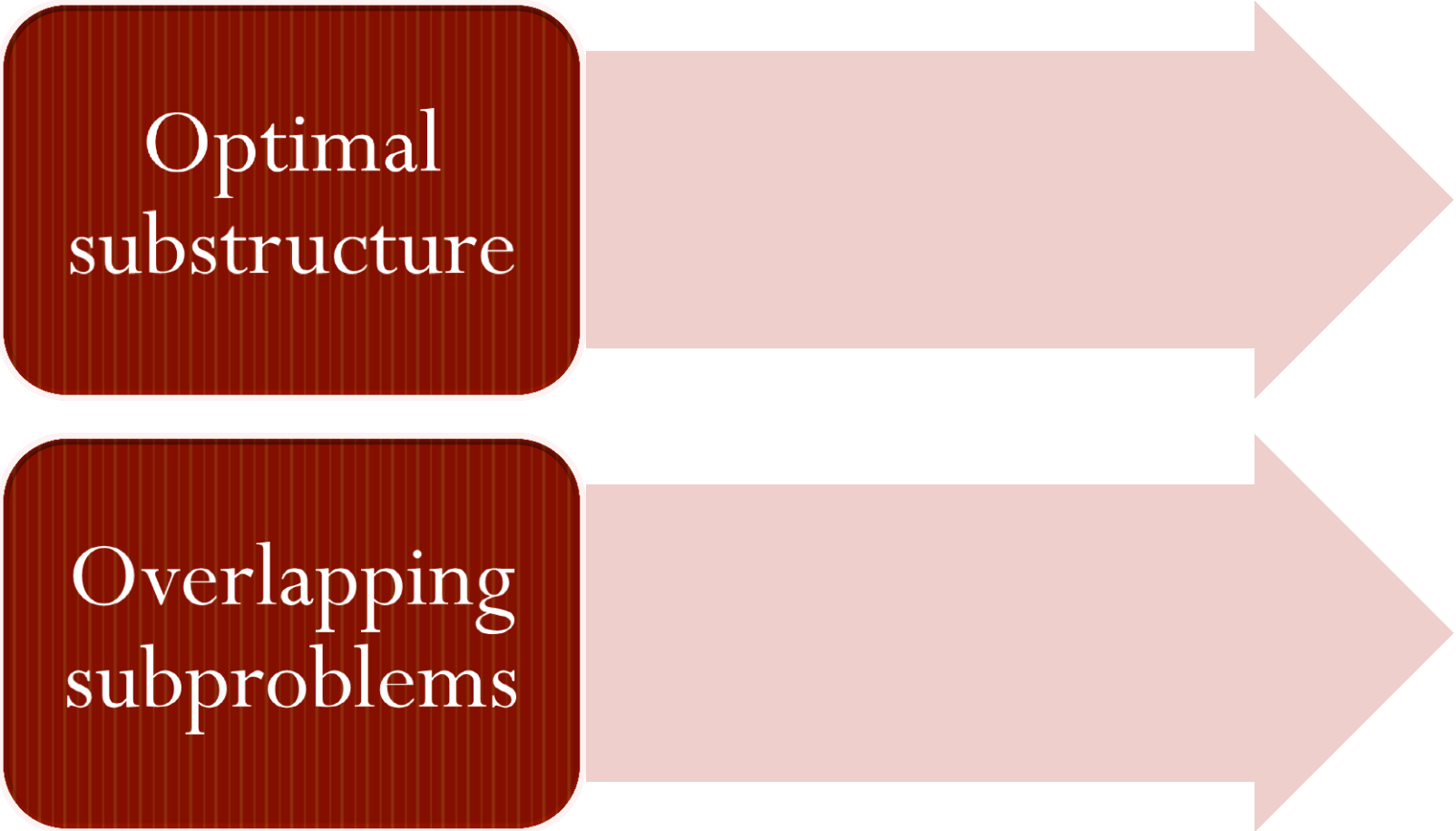
	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Dynamic Programming Applications

- Areas.
 - Bioinformatics.
 - Control theory.
 - Information theory.
 - Operations research.
 - Computer science: theory, graphics, **AI**, systems,
- Some famous dynamic programming algorithms.
 - **Viterbi** for hidden Markov models.
 - Unix *diff* for comparing two files.
 - Smith-Waterman for **sequence alignment**.
 - Bellman-Ford for **shortest path routing** in networks.
 - Cocke-Kasami-Younger for **parsing** context free grammars.

Elements of Dynamic Programming

Optimal
substructure



```
graph LR; A[Optimal substructure] --> B[ ]; B --> C[Overlapping subproblems]; C --> D[ ]
```

Overlapping
subproblems

Optimal Substructure

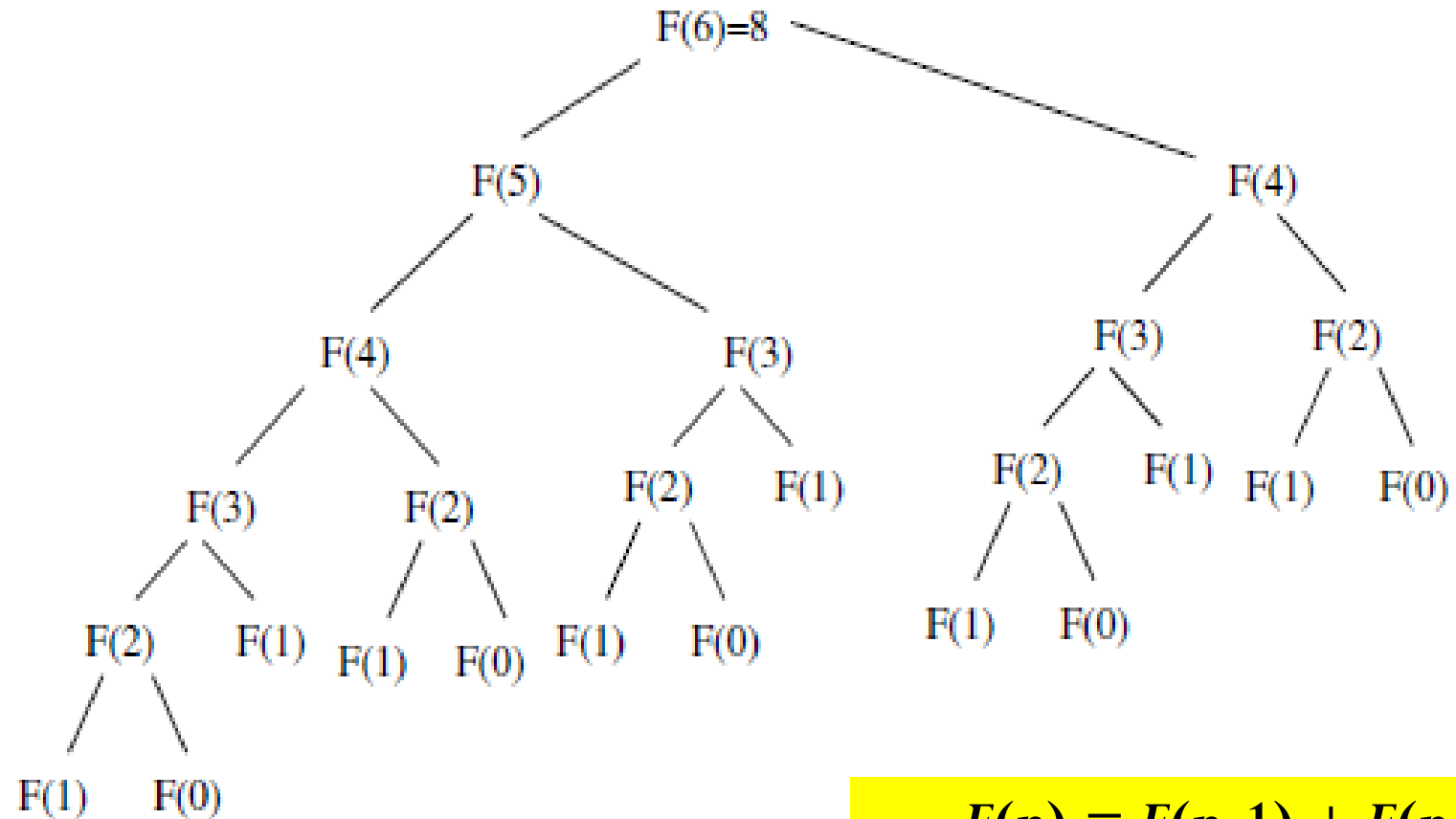
- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

Optimal Substructure

- Optimal substructure varies across problem domains:
 - 1. *How many subproblems* are used in an optimal solution.
 - 2. *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) \times (# of choices).
- Dynamic programming uses optimal substructure **bottom up**.
 - *First* find optimal solutions to **subproblems**.
 - *Then* choose which to use in optimal solution to the problem.

Nth Fibonacci number

- How many sub-problems and choices do the contain?

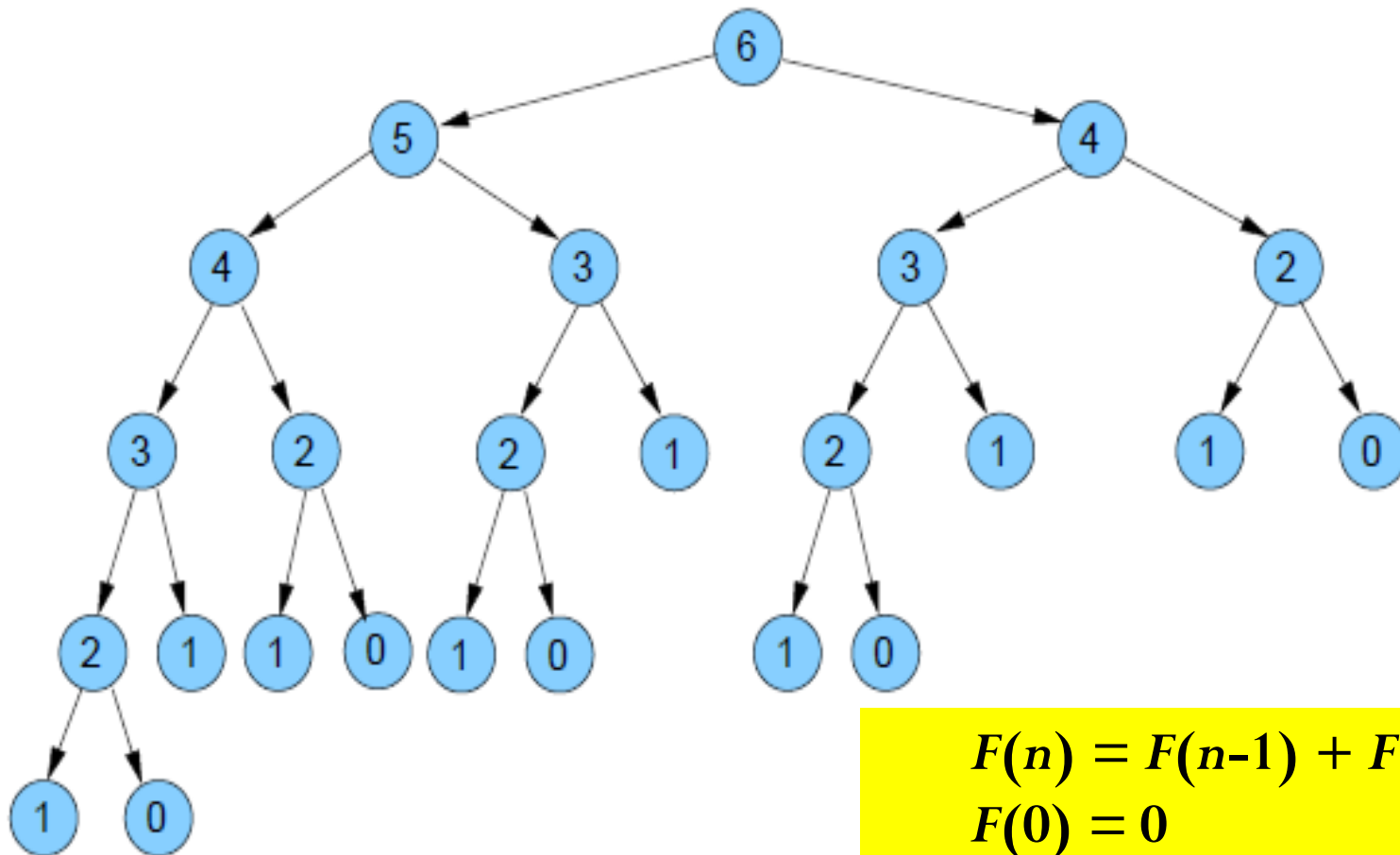


$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

Subproblem tree for Fibonacci function



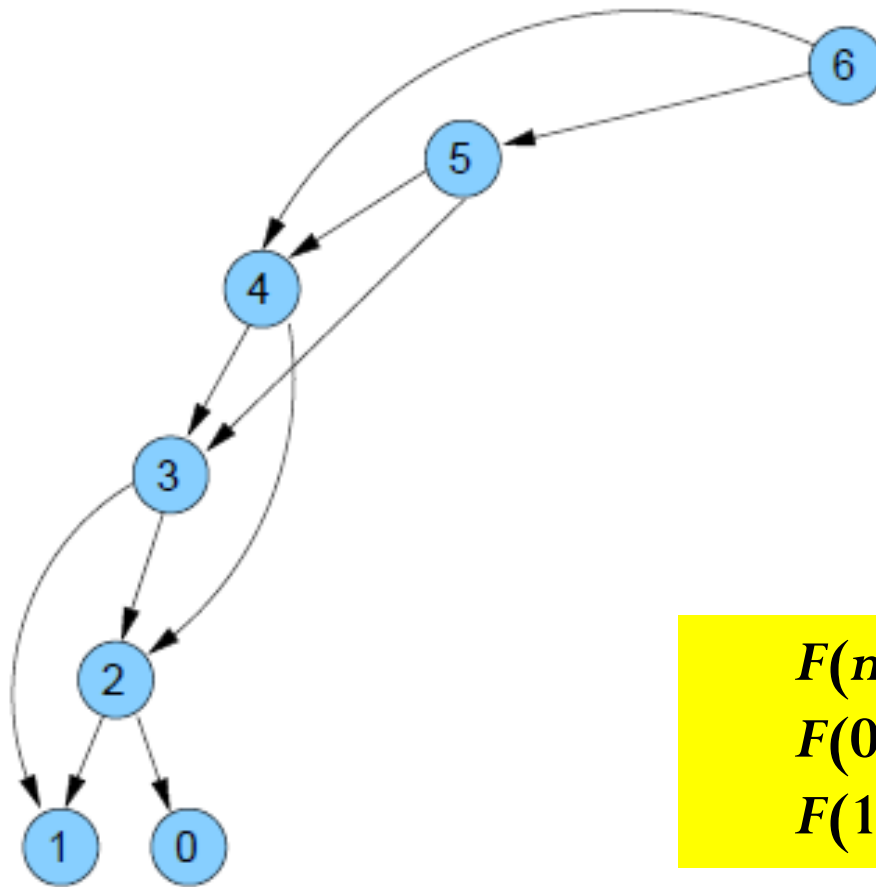
$$F(n) = F(n-1) + F(n-2)$$

$$F(\mathbf{0}) = \mathbf{0}$$

$$F(1) = 1$$

vertex labels are the parameters of the recursive calls

Subproblem graph for Fibonacci function

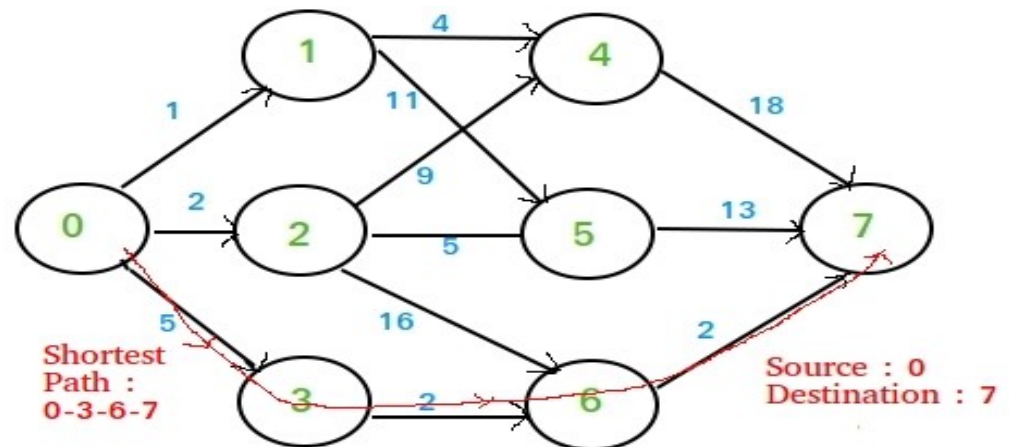


$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ F(0) &= 0 \\ F(1) &= 1 \end{aligned}$$

note that this is a dependency graph

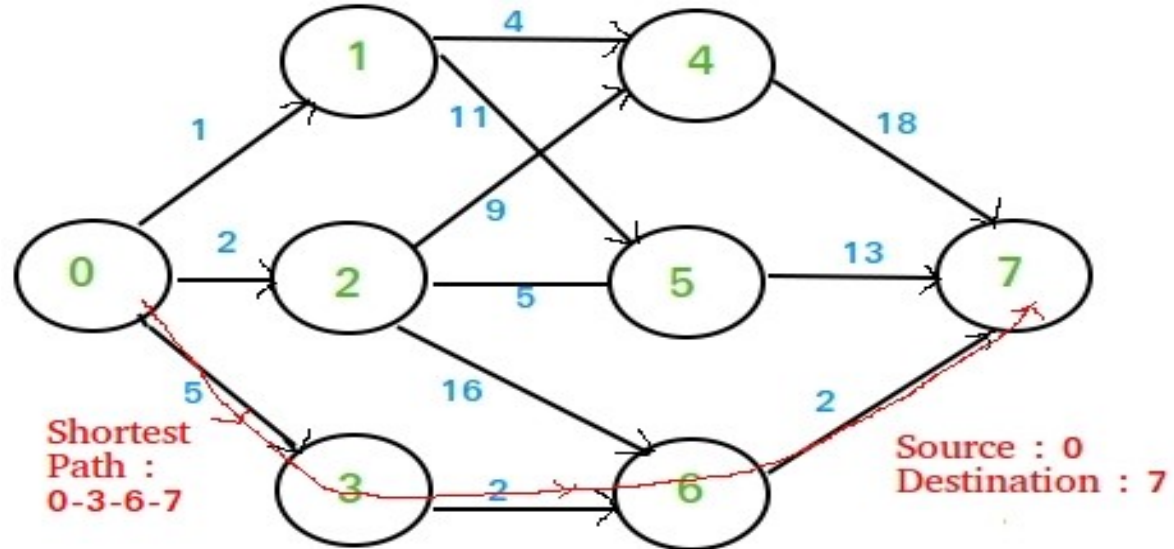
Optimal Substructure

- Does optimal substructure apply to all optimization problems? No.
- Applies to determining the shortest path but **NOT** the longest simple path of an unweighted directed graph.
- Why?
 - Shortest path has independent subproblems.
 - Solution to one subproblem does not affect solution to another subproblem of the same problem.
 - Subproblems are not independent in longest simple path.
 - Solution to one subproblem affects the solutions to other subproblems.
- Example:



Optimal Substructure

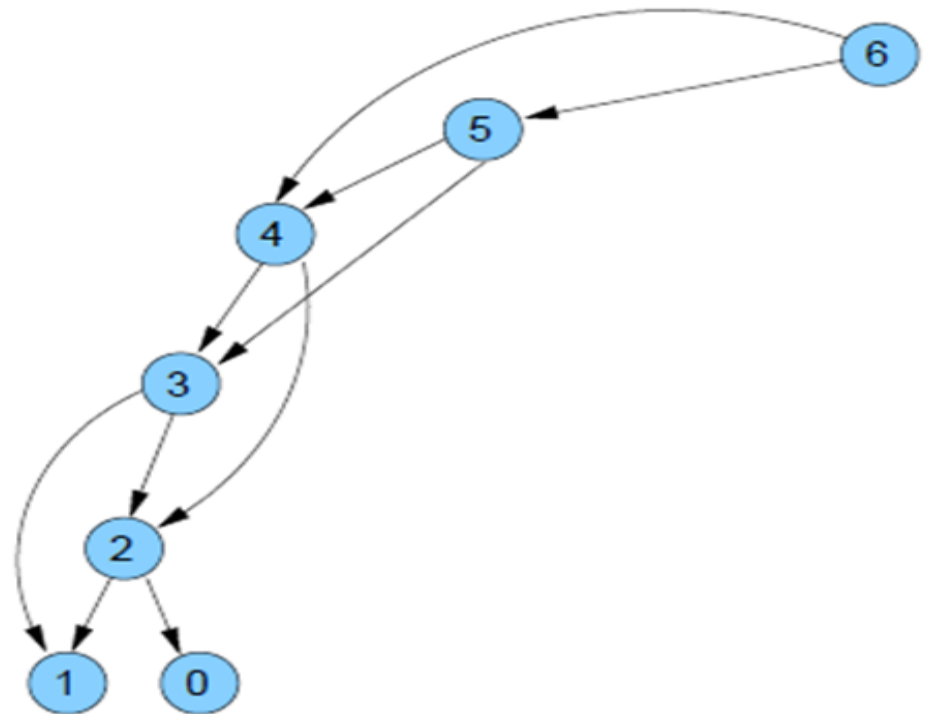
- Shortest path has independent subproblems.
- Solution to one subproblem does not affect solution to another subproblem of the same problem.
- Subproblems are not independent in longest simple path.
 - Solution to one subproblem affects the solutions to other subproblems.
- Example:



Overlapping Subproblems

The space of subproblems must be “small”.

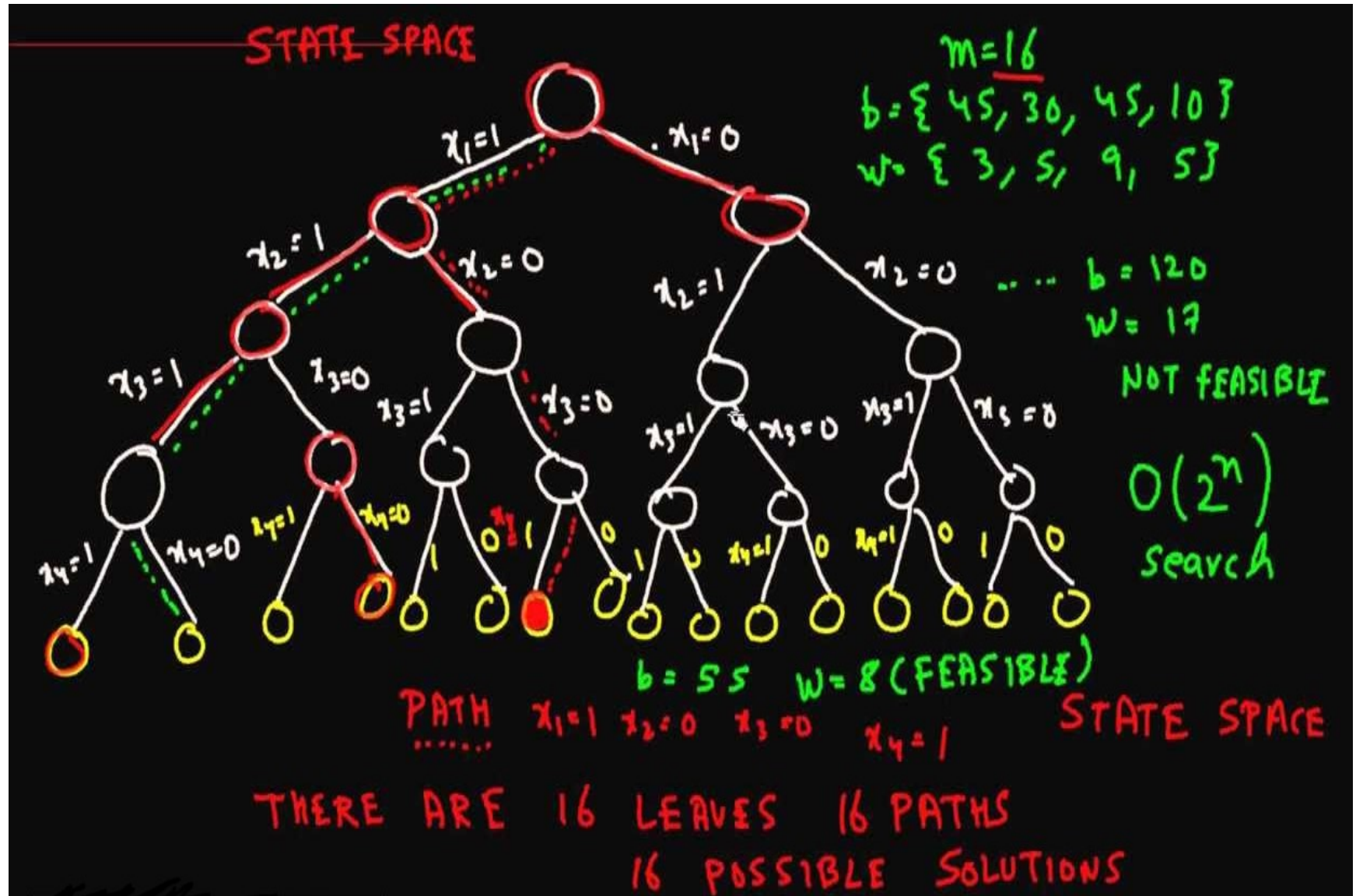
- The total number of distinct subproblems is a **polynomial** in the input size.
 - A recursive algorithm is exponential because it solves the same problems repeatedly.
 - If divide-and-conquer is applicable, then each problem solved will be brand new.



Dynamic programming recipe

1. Tackle the problem “top-down”; this yields a recursive algorithm
2. Define an appropriate dictionary; transform the recursive solution into a DP algorithm
3. Complexity of DP-algorithm is complexity **DFS** on subproblem graph
4. Choose an appropriate data structure for implementing the dictionary
5. If possible, analyse the subproblem graph and find a **reverse topological order**; **simplify** your DP-algorithm accordingly
6. Decide how to get the solution to the problem from the data in the dictionary

4 item 0/1 knapsack problem



7 Steps to solve a Dynamic Programming problem

1. How to recognize a DP problem
2. Identify problem variables
3. Clearly express the recurrence relation
4. Identify the base cases
5. Decide if you want to implement it iteratively or recursively
6. Add memorization
7. Determine time complexity

Divide-and-conquer vs Dynamic Programming

DIVIDE AND CONQUER

An algorithm that recursively breaks down a problem into two or more sub-problems of the same or related type until it becomes simple enough to be solved directly

Subproblems are independent of each other

Recursive

More time-consuming as it solves each subproblem independently

Less efficient

Used by merge sort, quicksort, and binary search

DYNAMIC PROGRAMMING

An algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property

Subproblems are interdependent

Non-recursive

Less time-consuming as it uses the answers of the previous subproblems

More efficient

Used by matrix chain multiplication, optimal binary search tree

Greedy vs Dynamic Programming

	Greedy Approach	Dynamic Programming
Main Concept	Choosing the best option that gives the best profit for the current step	Optimizing the recursive backtracking solution
Optimality	Only if we can prove that local optimality leads to global optimality	Gives an optimal solution
Time Complexity	Polynomial	Polynomial, but usually worse than the greedy approach
Memory Complexity	More efficient because we never look back to other options	Requires a DP table to store the answer of calculated states
Examples	Dijkstra and Prim's algorithm	0/1 Knapsack and Longest Increasing Subsequence

Greedy approach vs Dynamic programming

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the principle of optimality applies to the problem.

Greedy vs Divide & Conquer vs Dynamic Programming

Greedy	Divide & Conquer	Dynamic Programming
Optimises by making the best choice at the moment.	Optimises by breaking down a subproblem into simpler versions of itself and using multi-threading & recursion to solve.	Same as Divide and Conquer, but optimises by caching the answers to each subproblem as not to repeat the calculation twice.
Doesn't always find the optimal solution, but is very fast.	Always finds the optimal solution, but is slower than Greedy.	Always finds the optimal solution, but may be pointless on small datasets.
Requires almost no memory.	Requires some memory to remember recursive calls.	Requires a lot of memory for memoisation / tabulation

Recursive routine for n^{th} Fibonacci numbers

Recursive routine for n^{th} Fibonacci numbers

Definition of Fibonacci numbers:

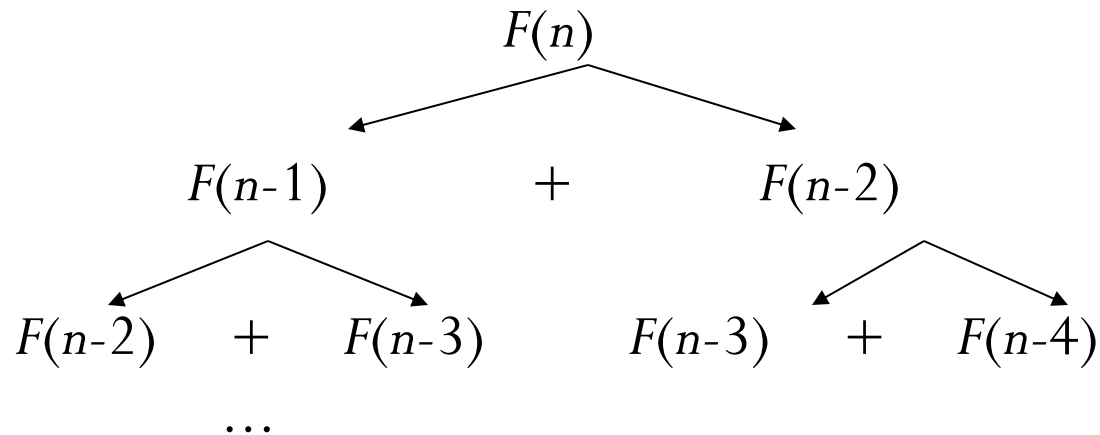
$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

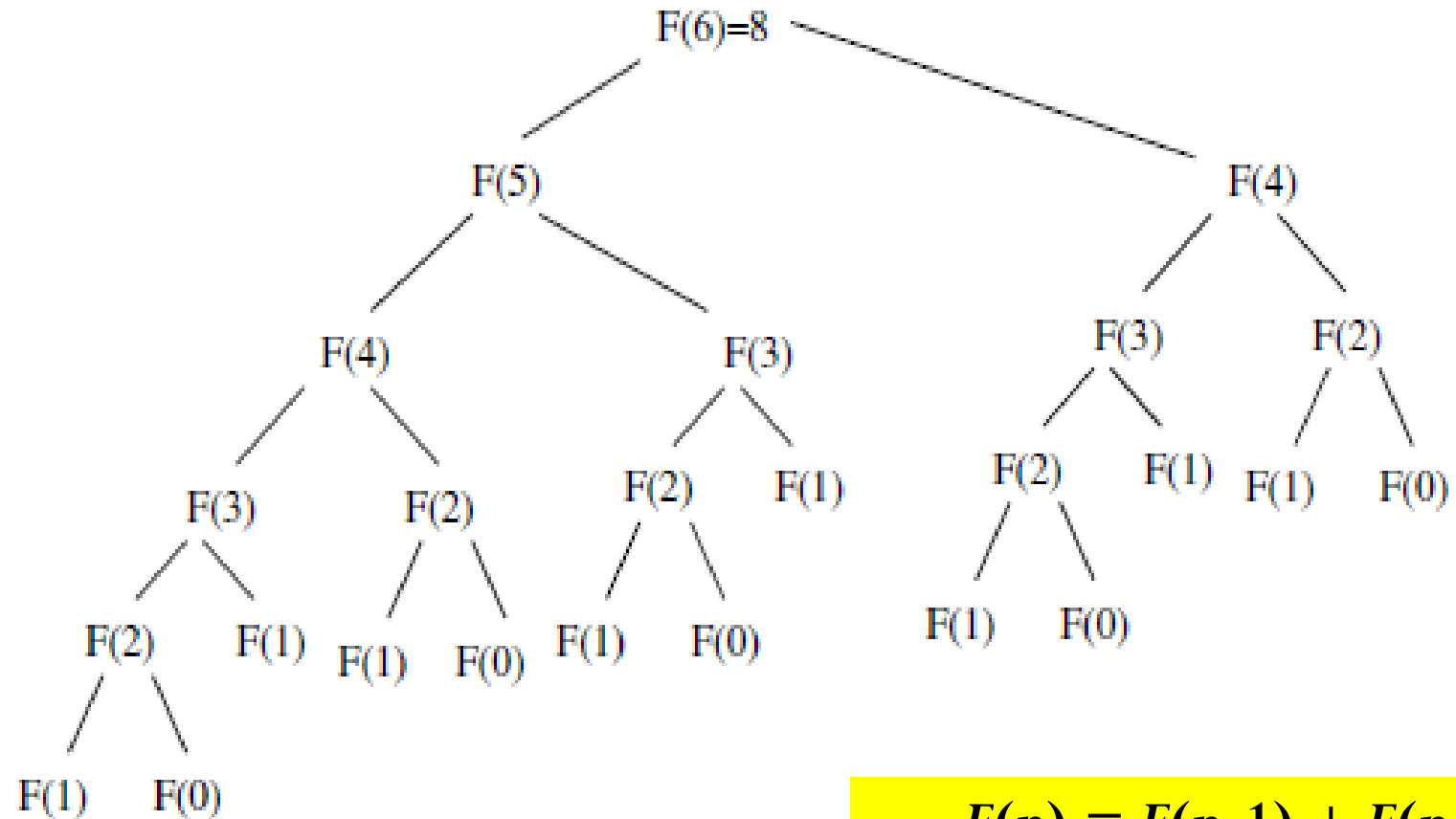
```
int Fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return Fib(n - 1) + Fib(n - 2);
}
```

Computing the n^{th} Fibonacci number recursively (top-down):



Nth Fibonacci number

- How many sub-problems and choices do the contain?

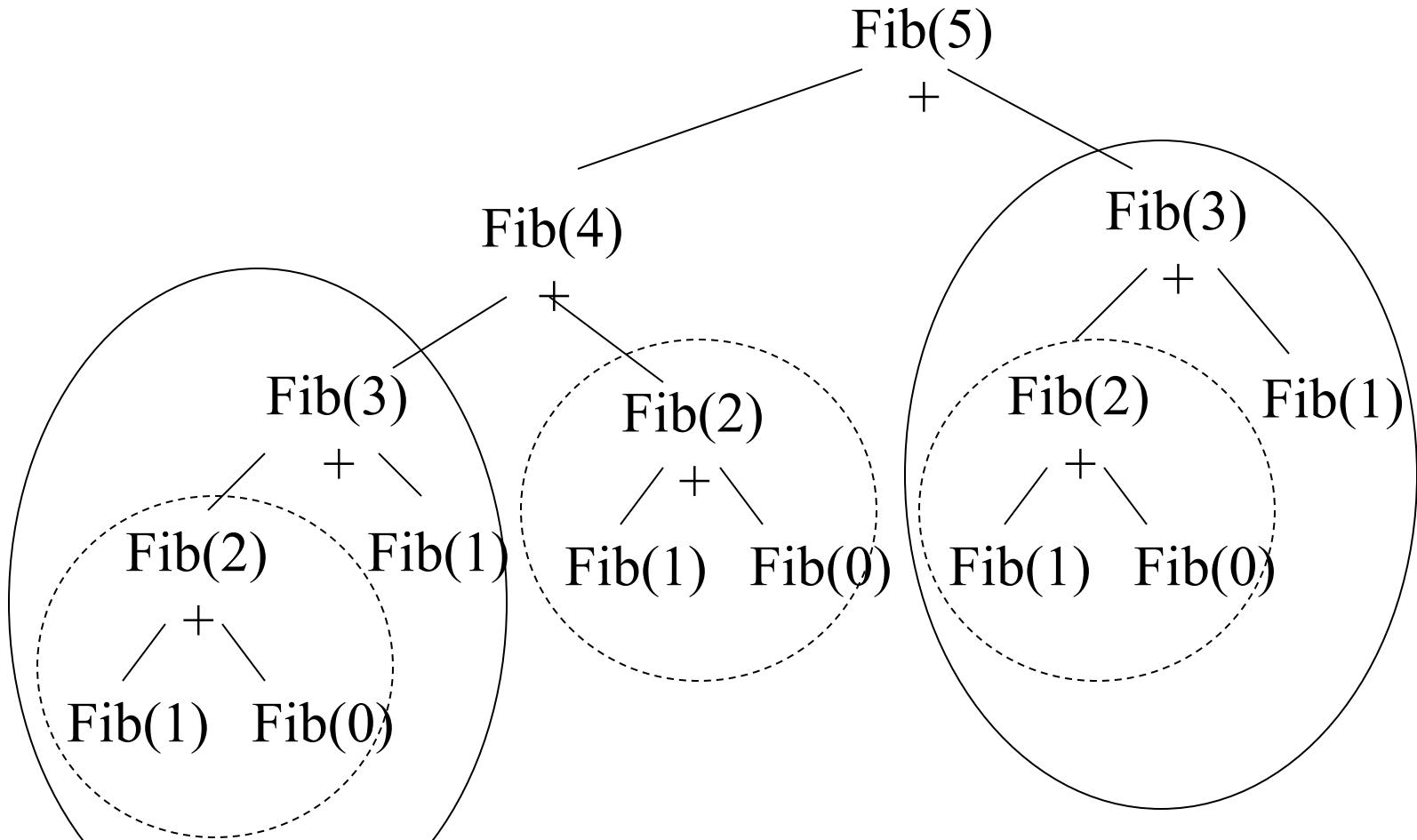


$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

The computation tree for computing Fibonacci numbers recursively



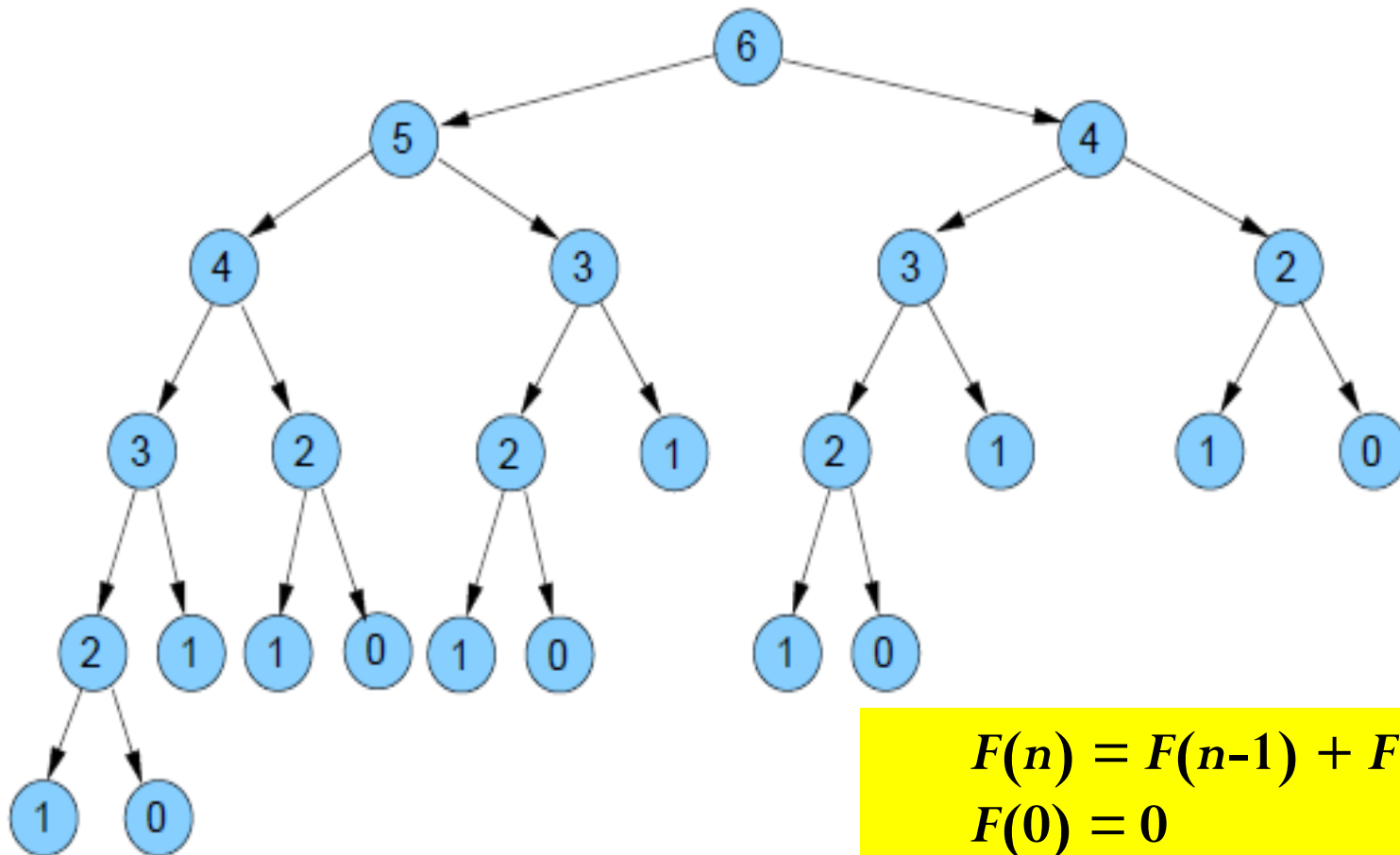
Complexity of Recursive routine for n^{th} Fibonacci numbers

- What is the Recurrence relationship?

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

- What is the solution to this?
- Space complexity $O(\log n)$
 - Clearly it is $O(2^n)$, but this is not tight.
 - A lower bound is $\Omega(2^{n/2})$.
 - You should notice that $T(n)$ grows very similarly to $F(n)$, so in fact $T(n) = \Theta(F(n))$.
- Obviously not very good, but we know that there is a better way to solve it!

Subproblem tree for Fibonacci function



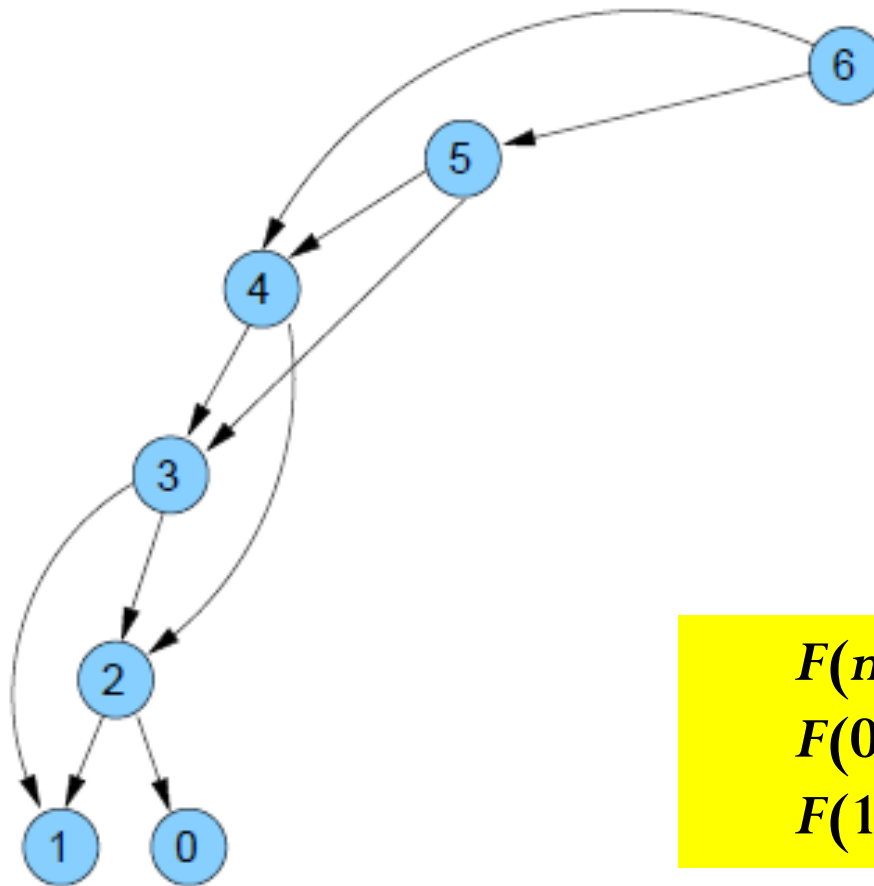
$$F(n) = F(n-1) + F(n-2)$$

$$F(\mathbf{0}) = \mathbf{0}$$

$$F(1) = 1$$

vertex labels are the parameters of the recursive calls

Subproblem graph for Fibonacci function



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

$$F(0) = 0$$

$$F(1) = 1$$

note that this is a dependency graph

Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = 1 + 0 = 1, \quad \dots, \quad F(n-2) = **, \quad F(n-1) = **$$

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

0 1 1 . . . $F(n-2)$ $F(n-1)$ $F(n)$



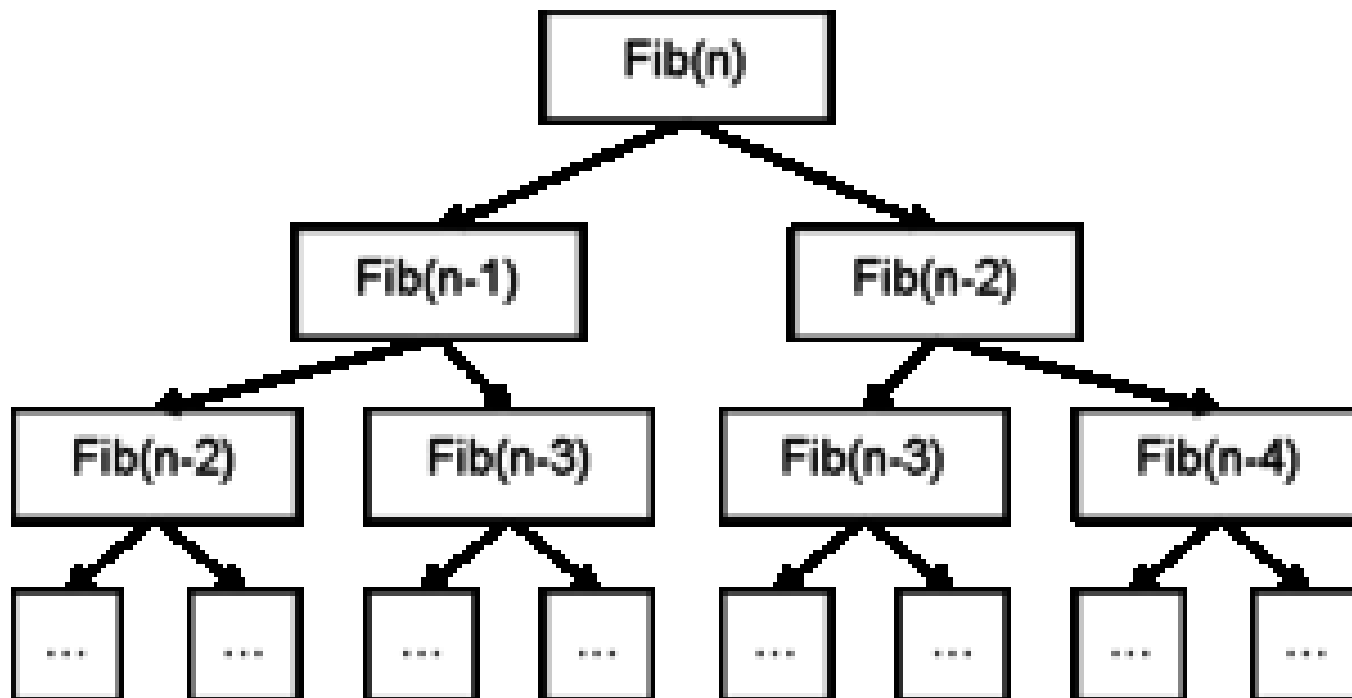
Efficiency:

Time – $O(n)$

Space – $O(n)$

Example: Fibonacci numbers

- The bottom-up approach is only $\Theta(n)$.
- Why is the top-down so inefficient?
 - Recomputed many sub-problems.
 - How many times is $F(n-5)$ computed?



Dynamic Programming Topics

- 501. The General Method
- 502. Multistage Graph
- 503. All-pairs Shortest Path
- 504. Single Source Shortest path
- 505. Optimal binary search trees
- 506. String editing
- 507. 0/1 knapsack problem
- 508. Reliability Design
- 509. The Travelling Salesperson Problem
- 510. Flow Shop scheduling
- 511. Matrix chain multiplication
- 512. Longest Common Sub sequence
- 513. Problem solving with dynamic programming

Dynamic programming : Conclusions

- Dynamic programming is a very useful technique for making a *sequence of interrelated decisions*.
- *It requires formulating an appropriate recursive relationship for each individual problem.* However, it provides a great computational savings over using exhaustive enumeration to find the best combination of decisions, especially for large problems.
- If a problem has 10 stages with 10 states and 10 possible decisions at each stage,
- then exhaustive enumeration must consider up to 10 billion combinations, whereas dynamic programming need make no more than a thousand calculations (10 for each state at each stage).
- A dynamic programming defined as the process for solving a problem by using solutions of sub-problems of smaller size.

Dynamic programming : Conclusions

- Dynamic programming is a general type of approach to problem solving, and the particular equations used must be developed to fit each situation.
- Therefore, a certain degree of ingenuity and insight into the general structure of dynamic programming problems is required to recognize when and how a problem can be solved by dynamic programming procedures.
- These abilities can best be developed by an exposure to a wide variety of dynamic programming applications and a study of the characteristics that are common to all these situations.
- Dynamic programming is a useful mathematical technique for making a sequence of interrelated decisions.
- Dynamic programming provides a systematic procedure for determining the optimal combination of decisions..

Dynamic programming: Conclusions

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be *recursively* described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memorization)
- Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results.



Thanks for Your Attention!

Exercises

Exercise

1. What is the principle of optimality that is used in dynamic programming paradigm? Explain with an example how the *use of table* is found to be efficient instead of using *recursion*.
2. Write an algorithm to find the minimum cost path from source to sink in a multistage graph using backward approach. What is the time complexity of the algorithm? How the time complexity of the algorithm is effected with their storage representation
3. One goal of global telecommunications is to enable data packets to reach their final intended destination as fast as possible. Describe an algorithm that could be used to generate the quickest route for packets over the network. The route should be traceable so it does not have to be re-calculated each time. Illustrate your answer with a simple worked example. Give diagrams where appropriate. Find the time complexity for your proposed solution

Exercise

4. Write a dynamic programming algorithm to compute a binomial coefficient $C(n,k)$? Prove that time complexity of the algorithm is **$O(nk)$** . What is the space complexity of this algorithm? Explain how the space efficiency of this algorithm can be improved.
5. What are the general characteristics of dynamic programming algorithm? What are the four basic steps to find optimal solution for the problems using dynamic programming? Compare divide-and-conquer and dynamic programming?
6. Write a dynamic programming algorithm to find a solution to 0-1 knapsack problem that yields maximum profit P for n items with the knapsack capacity W .

Exercise

7. Explain the principle of ordering Matrix multiplication in the light of dynamic programming. Show that the number of ways that n matrices can be multiplied is of $O(4^n/n^{3/2})$.
8. Coin Change - Given a value N , if we want to make change for N cents, and we have infinite supply of each of $S = \{S_1, S_2, \dots, S_m\}$ valued coins, how many ways can we make the change?
9. Longest Common Subsequence - Find the longest common subsequence of two strings A and B where the elements are letters from the two strings and they should be in the same order.
10. Longest Palindromic Subsequence - The question is same as above but the subsequence should be palindromic as well.
11. Minimum Number of Jumps - Given an array of integers where each element represents the maximum number of steps that can be made forward from that element, find the minimum number of jumps to reach the end of the array (starting from the first element).

The most popular dynamic programming problems

12. Given a matrix consisting of 0's and 1's, find the maximum size sub-matrix consisting of only 1's.
13. Given an array containing both positive and negative integers, find the contiguous array with the maximum sum.
14. Longest Increasing Subsequence - Find the length of the longest subsequence of a given sequence such that all the elements are sorted in increasing/non-decreasing order. There are many problems which reduce to the this problem such as box stacking and the building bridges. These days the interviewers expect an $(n \log n)$ solution.
15. Edit Distance - Given two strings and a set of operations Change (C), insert (I) and delete (D) , find minimum number of edits (operations) required to transform one string into another.

The most popular dynamic programming problems

16. 0/1 Knapsack Problem - A thief robbing a store and can carry a maximal weight of W into their knapsack. There are n items and i^{th} item weigh w_i and is worth v_i dollars. What items should thief take?
17. Balanced Partition - You have a set of n integers each in the range $0 \dots K$. Partition these integers into two subsets such that you minimize $|S1 - S2|$, where $S1$ and $S2$ denote the sums of the elements in each of the two subsets.
18. Write three different algorithm to compute n th Fibonacci number and comment on its time and space complexity.

Dynamic programming problems

19. **Fibonacci Sequence:** Write a dynamic programming solution to compute the n th Fibonacci number, ensuring your solution is optimized for time complexity.
20. **Coin Change Problem:** Given an unlimited supply of coins of given denominations, find the minimum number of coins required to make a specific amount of money.
21. **Longest Increasing Subsequence (LIS):** Given a sequence of numbers, find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.
22. **0/1 Knapsack Problem:** Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note that you cannot break an item, either pick the complete item or don't pick it (0/1 property).
23. **Matrix Chain Multiplication:** Given a chain of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.

Dynamic programming problems

- 24. **Edit Distance:** Given two strings `str1` and `str2`, and the operations insert, delete, and replace that can be performed on `str1`, find the minimum number of operations required to convert `str1` to `str2`.
- 25. **Longest Common Subsequence (LCS):** Given two sequences, find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
- 26. **Maximum Sum Increasing Subsequence:** Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order.
- 27. **Rod Cutting Problem:** Given a rod of length n and an array of prices that contains prices of all pieces of size smaller than n . Determine the maximum value obtainable by cutting up the rod and selling the pieces.
- 28. **Palindrome Partitioning:** Given a string, a partitioning of the string is a palindrome partitioning if every substring of the partition is a palindrome. Find the minimum number of cuts needed to perform a palindrome partitioning of the string

Solve this problem by dynamic programming

- A college student has 7 days remaining before final examinations begin in her four courses, and she wants to allocate this study time as effectively as possible. She needs at least 1 day on each course, and she likes to concentrate on just one course each day, so she wants to allocate 1, 2, 3, or 4 days to each course. Having recently taken an OR course, she decides to use dynamic programming to make these allocations to maximize the total grade points to be obtained from the four courses. She estimates that the alternative allocations for each course would yield the number of grade points shown in the following table:

Study Days	Estimated Grade Points			
	Course			
	1	2	3	4
1	3	5	2	6
2	5	5	4	7
3	6	6	7	9
4	7	9	8	9