Dynamic Programming and Applications

Mashiwat Tabassum Waisy

Lecturer

Department of CSE

STAMFORD UNIVERSITY BANGLADESH

Dynamic Programming

D*ynamic Programming* is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

- “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

Divide-and-conquer

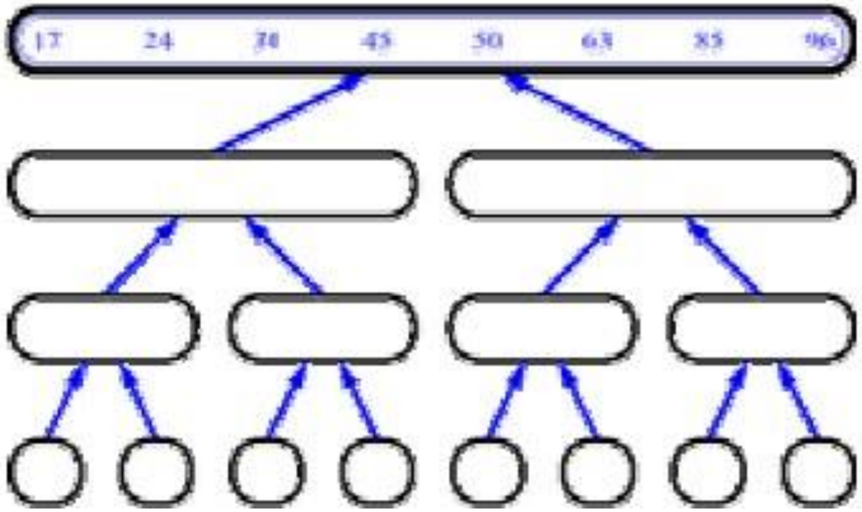
- **Divide-and-conquer** method for algorithm design:
- **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
- **Conquer**: conquer recursively to solve the subproblems
- **Combine**: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem

Divide-and-conquer - Example

- For example, **MergeSort**

```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

- The **subproblems** are **independent**, **all different.**



Dynamic programming

- **Dynamic programming** is a way of improving on **inefficient** *divide-*
and-conquer algorithms.

- By "*inefficient*", we mean that ***the same recursive call is made over and over***.

- If **same** *subproblem* is solved several times, we can use **table** to store result of a subproblem the first time it is computed and thus never have to recompute it again.

- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share subsubproblems.

- **"Programming"** refers to a tabular method

Difference between DP and Divide- and-Conquer

- Using Divide-and-Conquer to solve these problems is **inefficient** because the **same** common **subproblems** have to be solved **many times**.

- DP will solve each of them **once** and **their answers are stored in a table** for future use.

Dynamic Programming VS. Divide-and Conquer

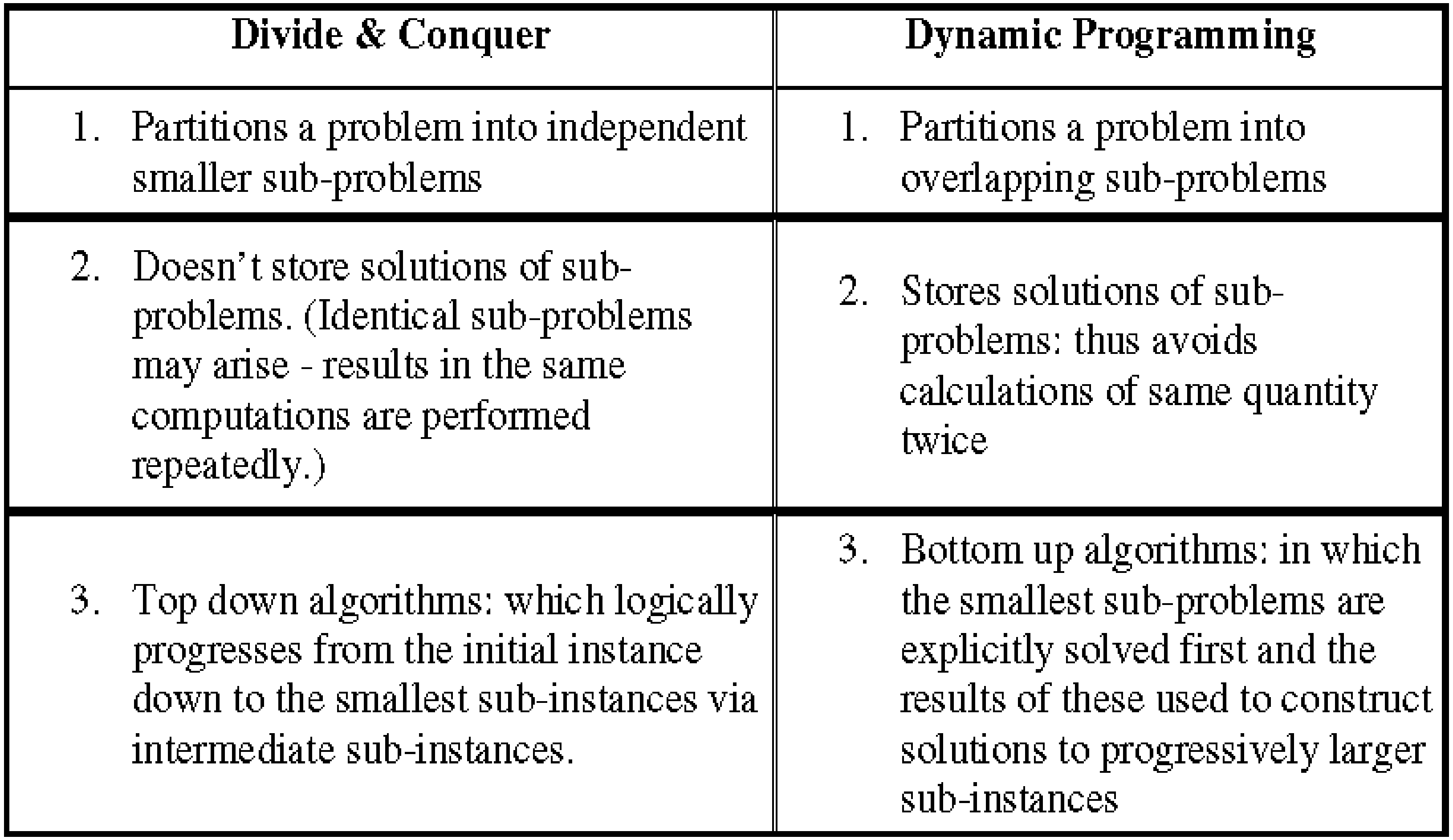
Divide-and-Conquer

- partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem

Dynamic Programming

- applicable when the subproblems are not independent, that is, when subproblems share subproblems.
- Solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered

Dynamic Programming vs. Recursion and Divide & Conquer



Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Greedy Method v/s Dynamic Programing

- Both are used for Optimization problem. It required minimum or maximum results.
- In Greedy Method:- Always follow a function until you find the optimal result whether it is maximum or minimum.
- Prims algorithm for MST
- Dijkstra Algorithm for finding shortest path.

Greedy Method v/s Dynamic Programing

- In Dynamic programing we will try to find all possible solution and
  then we'll pick the best solution which is optimal.
- It is time consuming as compared to greedy method.
- It use recursive formulas for solving problem.
- It follows the principal of optimality.

Dynamic Programming VS. Greedy Method

Greedy Method

- only one decision sequence is ever genertated.

Dynamic Programming

- Many decision sequences may be genertated.

Greedy vs. Dynamic Programming

Greedy Technique	Dynamic Technique
• It gives local “Optimal Solution”	• It gives “Global Optimal Solution”
• Print in time makes “local optimization”	• It makes decision “ smart recursion ”
• More efficient as compared to dynamic programing	• Less efficient as compared too greedy technique
• Example:- Fractional Knapsack	• Example:- 0/1 Knapsack

Principle of Optimality

- Principle of optimality says that a problem can be solved by sequence of decisions to get the optimal solution.
- It follows Memoization Problem
- An optimal sequence of decisions has the property that whatever the initial state an decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- Essentially, this principles states that the optimal solution for a larger subproblem contains an optimal solution for a smaller subproblem.

Elements of Dynamic Programming (DP)

DP is used to solve problems with the following characteristics:

- 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 **optimal solution** to the problem contains within **optimal solutions to** its **subproblems**.
- Overlapping sub-problems
  - there exist some places where we solve the **same subproblem** more than **once**.

Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal substructure

2. Recursively define the value of an optimal solution

3. Compute the value bottom up

4. (if needed) Construct an optimal solution

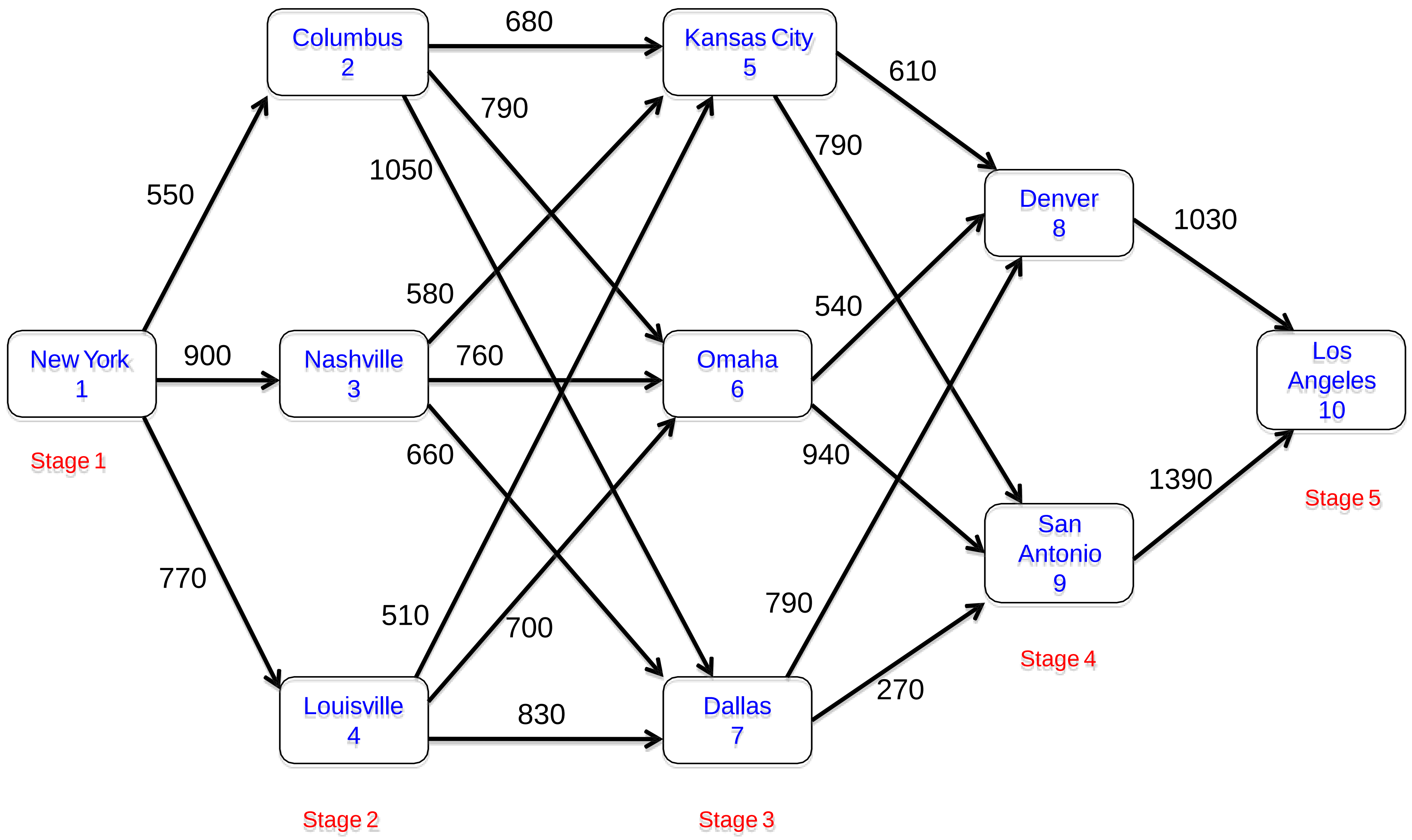
Principle of Optimality

- The dynamic Programming works on a principle of optimality.
- Principle of optimality states that in an optimal sequence of decisions or choices, each sub sequences must also be optimal.

A typical example: Shortest Path

- Ben plans to drive from NY to LA
- Has friends in several cities
- After 1 day's driving can reach Columbus, Nashville, or Louisville
- After 2 days of driving can reach Kansas City, Omaha, or Dallas
- After 3 days of driving can reach Denver or San Antonio
- After 4 days of driving can reach Los Angeles
- The actual mileages between cities are given in the figure (next slide)
- Where should Ben spend each night of the trip to minimize the number of miles traveled?

Shortest Path: network figure



Shortest Path problem: Solution

- The problem is solved **recursively by working backward** in the network
- Let c_{ij} be the mileage between cities i and j
- Let $f_t(i)$ be the length of the shortest path from city i to LA (city i is in stage t)

Stage 4 computations are obvious:

- $f_4(8) = 1030$
- $f_4(9) = 1390$

Stage 3 computations

Work backward one stage (to stage 3 cities) and find the shortest path to LA from each stage 3 city.

To determine $f_3(5)$, note that the shortest path from city 5 to LA must be one of the following:

- *Path 1*: Go from city 5 to city 8 and then take the shortest path from city 8 to city 10.
- *Path 2*: Go from city 5 to city 9 and then take the shortest path from city 9 to city 10.

$$f_3(5) = \min \begin{cases} c_{58} + f_4(8) = 610 + 1030 = 1640^* \\ c_{59} + f_4(9) = 790 + 1390 = 2180 \end{cases}$$

Similarly, $$f_3(6) = \min \begin{cases} c_{68} + f_4(8) = 540 + 1030 = 1570^* \\ c_{69} + f_4(9) = 940 + 1390 = 2330 \end{cases}$$

$$f_3(7) = \min \begin{cases} c_{78} + f_4(8) = 790 + 1030 = 1820 \\ c_{79} + f_4(9) = 270 + 1390 = 1660^* \end{cases}$$

Stage 2 computations

Work backward one stage (to stage 2 cities) and find the shortest path to LA from each stage 2 city.

$$f_2(2) = \min \begin{cases} c_{25} + f_3(5) = 680 + 1640 = 2320^* \\ c_{26} + f_3(6) = 790 + 1570 = 2360 \\ c_{27} + f_3(7) = 1050 + 1660 = 2710 \end{cases}$$

$$f_2(3) = \min \begin{cases} c_{35} + f_3(5) = 580 + 1640 = 2220^* \\ c_{36} + f_3(6) = 760 + 1570 = 2330 \\ c_{37} + f_3(7) = 660 + 1660 = 2320 \end{cases}$$

$$f_2(4) = \min \begin{cases} c_{45} + f_3(5) = 510 + 1640 = 2150^* \\ c_{46} + f_3(6) = 700 + 1570 = 2270 \\ c_{47} + f_3(7) = 830 + 1660 = 2490 \end{cases}$$

Stage 1 computations

Now we can find $f_1(1)$, and the shortest path from NY to LA.

$$f_1(1) = \min \begin{cases} c_{12} + f_2(2) = 550 + 2320 = 2870^* \\ c_{13} + f_2(3) = 900 + 2220 = 3120 \\ c_{14} + f_2(4) = 770 + 2150 = 2920 \end{cases}$$

Checking back our calculations, the shortest path is

1 – 2 – 5 – 8 – 10

that is,

NY – Columbus – Kansas City – Denver – LA

with total mileage 2870.

General characteristics of Dynamic Programming

- The problem structure is divided into **stages**
- Each stage has a number of **states** associated with it
- Making **decisions** at one stage transforms one state of the current stage into a state in the next stage.
- Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. This is known as the **principle of optimality** for dynamic programming.
- The principle of optimality allows to solve the problem stage by stage **recursively**.

Division into stages

The problem is divided into smaller subproblems each of them represented by a stage.

The stages are defined in many different ways depending on the context of the problem.

If the problem is about **long-time development of a system** then the stages naturally correspond to **time periods**.

If the goal of the problem is to **move some objects from one location to another on a map** then partitioning the map into several **geographical regions** might be the natural division into stages.

Generally, if an accomplishment of a certain task can be considered as a **multi-step process** then each stage can be defined as a step in the process.

States

Each stage has a number of *states* associated with it. Depending what decisions are made in one stage, the system might end up in different states in the next stage.

If a geographical region corresponds to a stage then the states associated with it could be some particular locations (cities, warehouses, etc.) in that region.

In other situations a state might correspond to amounts of certain resources which are essential for optimizing the system.

Decisions

Making *decisions* at one stage transforms one state of the current stage into a state in the next stage.

In a geographical example, it could be a decision to go from one city to another.

In resource allocation problems, it might be a decision to create or spend a certain amount of a resource.

For example, in the shortest path problem three different decisions are possible to make at the state corresponding to Columbus; these decisions correspond to the three arrows going from Columbus to the three states (cities) of the next stage: Kansas City, Omaha, and Dallas.

Optimal Policy and Principle of Optimality

The goal of the solution procedure is to find an **optimal policy** for the overall problem, i.e., an optimal policy decision at each stage for each of the possible states.

Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. This is known as the **principle of optimality** for dynamic programming.

For example, in the geographical setting the principle works as follows: the optimal route from a current city to the final destination does not depend on the way we got to the city.

A system can be formulated as a dynamic programming problem only if the principle of optimality holds for it.

Recursive solution to the problem

The principle of optimality allows to solve the problem stage by stage recursively.

The solution procedure first finds the *optimal policy for the last stage*. The solution for the last stage is normally trivial.

Then a **recursive relationship** is established which identifies the optimal policy for stage t, given that stage t+1 has already been solved.

When the recursive relationship is used, the solution procedure starts at the end and moves *backward* stage by stage until it finds the optimal policy starting at the *initial* stage.

Example 1: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- **Computing the *n*th Fibonacci number recursively (top-down):**

$$\begin{tikzcd}
 & & & F(n) \arrow[dll] \arrow[drr] & & & \\
 & F(n\text{-}1) \arrow[dl] \arrow[dr] & & + & & F(n\text{-}2) \arrow[dl] \arrow[dr] & \\
F(n\text{-}2) & + & F(n\text{-}3) & & F(n\text{-}3) & + & F(n\text{-}4)
\end{tikzcd}$$

•••

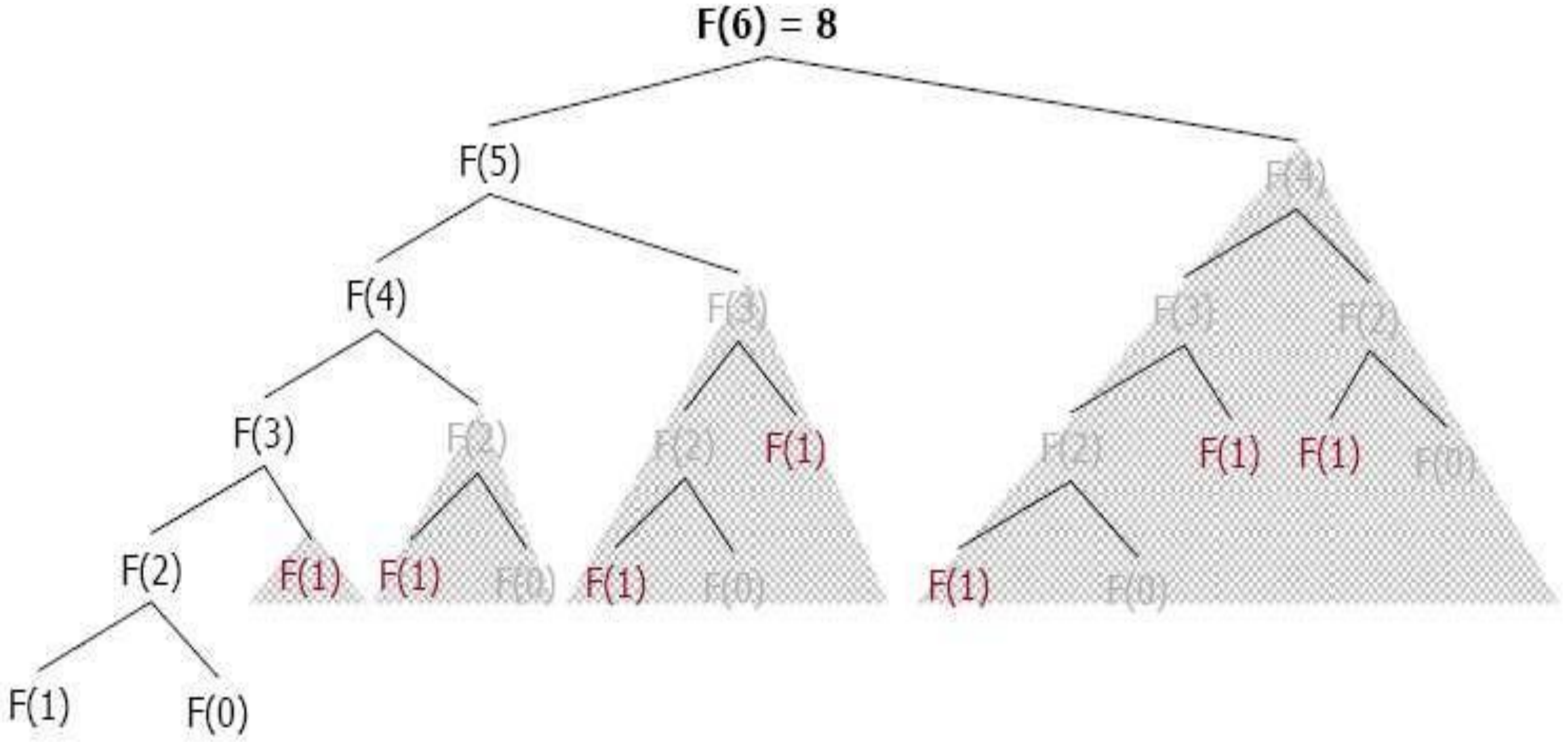
Fibonacci Numbers

- *Fn= Fn-1+ Fn-2* $n \geq 2$
- *F0 =0, F1 =1*
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- Straightforward recursive procedure is slow!

- Let’s draw the recursion tree

Fibonacci Numbers



- We keep calculating the same value over and over!

Fibonacci Numbers

- We can calculate *Fn* in ***linear* time** by remembering solutions to the solved subproblems – *dynamic programming*
- Compute solution in a bottom-up fashion
- In this case, only two values need to be remembered at any time

```
Fibonacci(n)
    $F_0 \leftarrow 0$
    $F_1 \leftarrow 1$
    for $i \leftarrow 2$ to $n$ do
        $F_i \leftarrow F_{i-1} + F_{i-2}$
```

Applications

- Matrix Chain Multiplication(MCM)
- Optimal Binary Search Tree
- All Pairs Shortest Path Problems
- Travelling Salesperson Problem(TSP)
- 0/1 Knapsack Problem
- Reliability Design
- Longest Common Subsequences(LCS)

Advantages

1) The process of breaking down a complex problem into a series of interrelated sub problems often provides insight into the nature of problem
2) Because dynamic programming is an approach to optimization rather than a technique it has flexibility that allows application to other types of mathematical programming problems
3) The computational procedure in dynamic programming allows for a built in form of sensitivity analysis based on state variables and on variables represented by stages
4) Dynamic programming achieves computational savings over complete enumeration.

Disadvantages

1)More expertise is required in solving dynamic programming problem then using other methods

2)Lack of general algorithm like the simplex method. It restricts computer codes necessary for inexpensive and widespread use

3)the biggest problem is dimensionality. This problems occurs when a particular application is characterized by multiple states. It creates lot of problem for computers capabilities & is time consuming

Thank you