BITS, PILANI – K. K. BIRLA GOA CAMPUS

Design & Analysis of Algorithms

(CS F364)

Lecture No. 3

Course Objective 1:

Algorithm Design Techniques

We already know one popular strategy

Divide & Conquer

Consider the **Coin Change Problem** with coins of denomination 1, 5, 10 & 25

Solution is Easy

What is the guarantee that solution works!

Introduce two more popular & widely applicable problem solving strategies:

- **Dynamic Programming**
- **Greedy Algorithms**

Key Questions

Course Objective 2:

One of the objectives of this course is to look at Question 5 to Question 8 in detail for a class of problems

Understand famous **P vs NP** problem

We strongly believe that certain important class of problems will not have polynomial time solution.

Course Objective - 3

- How to deal with the class of problems for which we strongly believe that no polynomial time algorithm will exist?
- This class consists of important practical problems

Example: Traveling Salesman Problem, 0-1 Knapsack Problem, Bin Packing Problem

And Many more

Course Objective – 3

For a certain class of problems to study, develop and analyze **'good' approximation algorithms**.

Course Objective - 4

Course Objective – 4

Study type of binary search trees called **Treap** for which **expected number of rotations needed for balancing is constant**

We will be proving that **expected number of rotations needed for balancing in Treap is 2**

(something really strong)

Treap is an Randomized Data Structure

Algorithm Design Strategies-Review

Algorithm Design Strategies

- **Divide & Conquer**

Algorithm Design Strategies

- **Divide & Conquer**
  - binary search
  - merge sort
  - quick sort
  - Matrix Multiplication (Strassen Algorithm)

Many More

Many standard iterative algorithms can be written as Divide & Conquer Algorithms.

Example: Sum/Max of n numbers

Question: Any advantage in doing so?

Divide & Conquer

Divide-and-conquer algorithms:

1. Dividing the problem into **smaller sub-problems**

   (**independent sub-problems**)

2. Solving those sub-problems

3. Combining the solutions for those smaller sub-problems to solve the original problem

Divide & Conquer

How to analyze Divide & Conquer Algorithms

Generally, using **Recurrence Relations**

- Let T(n) be the number of operations required to solve the problem of size n.

Then T(n) = **a** T(**n/b**) + **c(n)** where

- each sub-problem is of size **n/b**
- There are **a** such sub-problems
- **c(n)** extra operations are required to combine the solutions of sub-problems into a solution of the original problem

Divide & Conquer

- How to solve Recurrence Relations

- **Substitution Method**
  - works in simple relations
- **Masters Theorem**
  - See Text Book for details

Text Book

- **Text Book**

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein

Introduction to Algorithms

Third Edition, PHI Learning Private Limited, 2015

Dynamic Programming

- **D*ynamic Programming*** is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

  i.e, Applicable when subproblems are not independent

  i.e., Subproblems share subsubproblems

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems

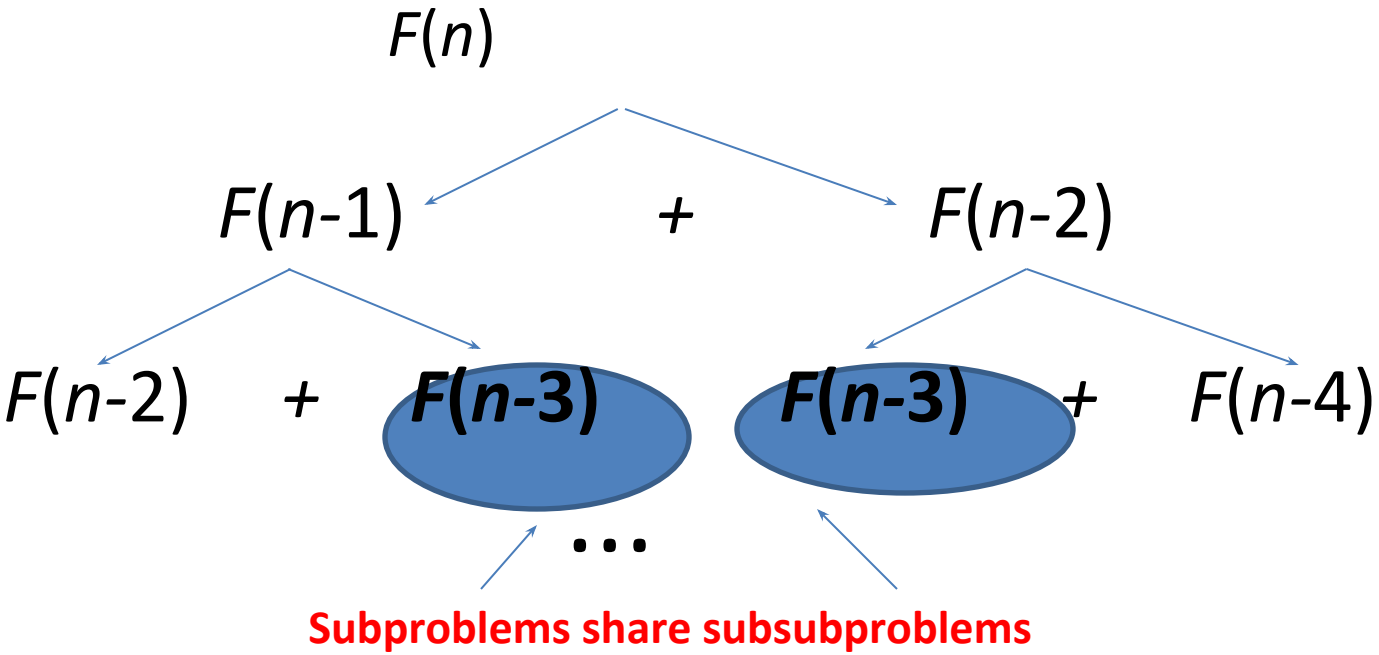
Dynamic Programming

- **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

Example: Fibonacci numbers

- **Fibonacci numbers:**
  F(n) = F(n-1) + F(n-2)
  F(0) = 0 & F(1) = 1

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

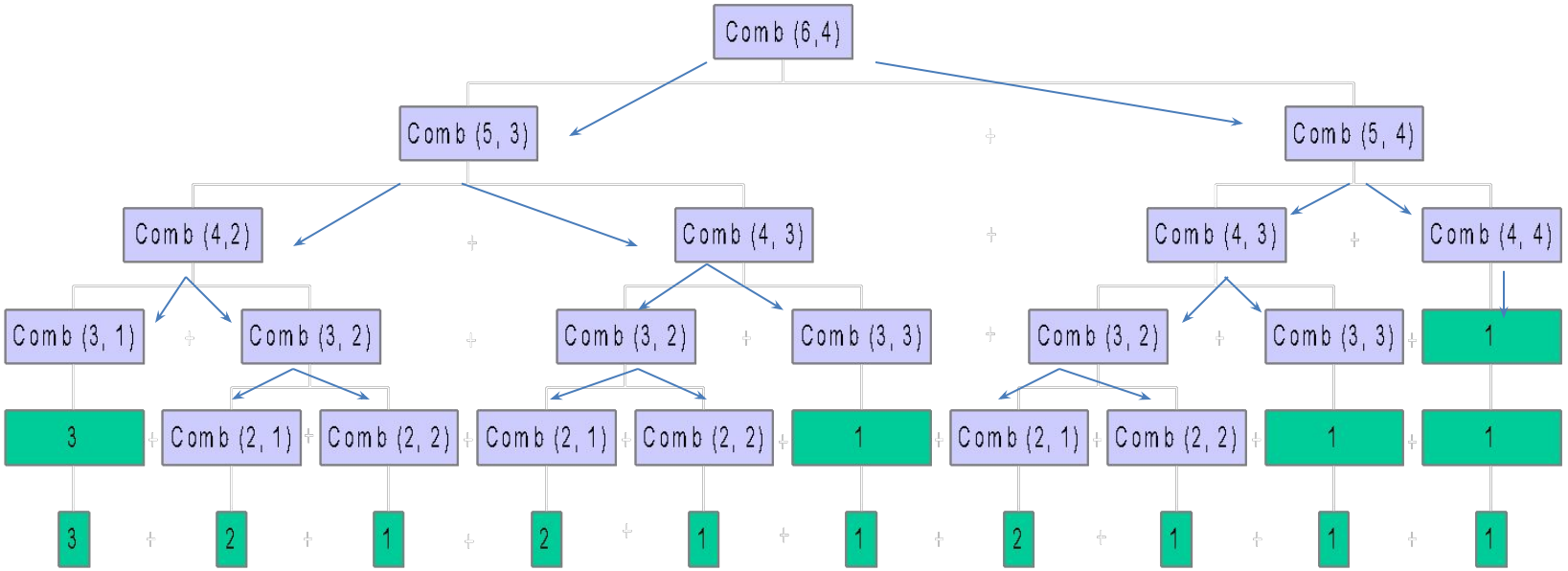


Computing binomial coefficient

Recurrence:

$$C(n, k) = C(n-1, k) + C(n-1, k-1) \quad \text{for } n > k > 0$$

$$C(n, 0) = 1, \quad C(n, n) = 1 \quad \text{for } n \geq 0$$



Computing binomial coefficient

Value of ***C(n,k)*** can be computed by filling a table:

0	0	1	2	. . .	*k*-1	*k*
0	1					
1	1	1				
.						
.						
.						
n-1					*C*(*n*-1,*k*-1)	*C*(*n*-1,*k*)
n					*C*(*n*,*k*)	

Time Complexity
O(nk)