Syllabus

UNIT I **INTRODUCTION:** Introduction to Algorithms – Fundamentals of Algorithmic Problem Solving – Fundamentals of the Analysis of Algorithmic Efficiency – Analysis Framework – Asymptotic Notations and Basic Efficiency Classes – Mathematical Analysis of Recursive Algorithms – Mathematical Analysis of Non-recursive Algorithms

UNIT II **DIVIDE-AND-CONQUER:** Divide and Conquer Methodology – Binary Search – Merge Sort – Quick Sort – Heap Sort – Multiplication of Large Integers – Strassen's Matrix Multiplication

UNIT III **DYNAMIC PROGRAMMING:** Dynamic Programming – Change-making Problem – Computing a Binomial Coefficient – All-pairs Shortest-paths Problem – Warshall's and Floyd's Algorithms – 0/1 Knapsack Problem

UNIT IV **GREEDY TECHNIQUE:** Greedy Technique – Minimum Spanning Tree – Prim's Algorithm – Kruskal's Algorithm – Single-source Shortest-paths Problem – Dijkstra's Algorithm – Huffman Coding – Fractional Knapsack Problem

UNIT V **BACKTRACKING AND BRANCH-AND-BOUND:** Backtracking – N-Queens Problem – Hamiltonian Circuit Problem – Subset Sum Problem – Branch-and- Bound – Travelling Salesman Problem

UNIT VI **LIMITATIONS OF ALGORITHM POWER:** P and NP Problems – NP-Complete Problems – Decision Trees – Information Retrieval – Pattern Matching – Data Science Algorithms

UNIT III DYNAMIC PROGRAMMING:

Dynamic Programming – Change-making Problem – Computing a Binomial Coefficient – All-pairs Shortest-paths Problem – Warshall's and Floyd's Algorithms – 0/1 Knapsack Problem

Dynamic Programming

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

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

- "Programming" here means "planning"

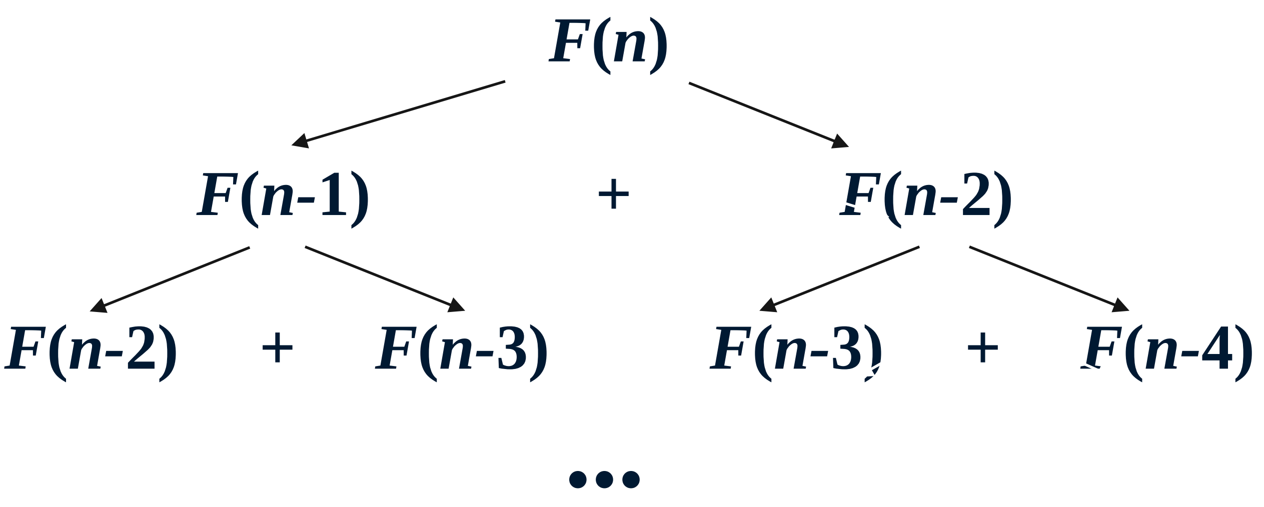
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Example: 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):**



Example: Fibonacci numbers (cont.)

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

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

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

Efficiency:
- time
- space

What if we solve it recursively?

Examples of DP algorithms

- Computing a binomial coefficient
- Longest common subsequence
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack

Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \dots + C(n,k)a^{n-k}b^k + \dots + C(n,n)a^0 b^n$$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

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

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

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

Computing $C(n,k)$: pseudocode and analysis

```
ALGORITHM Binomial(n, k)
    //Computes C(n, k) by the dynamic programming algorithm
    //Input: A pair of nonnegative integers n ≥ k ≥ 0
    //Output: The value of C(n, k)
    for i ← 0 to n do
        for j ← 0 to min(i, k) do
            if j = 0 or j = i
                C[i, j] ← 1
            else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
    return C[n, k]
```

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Computing $C(n,k)$: pseudocode and analysis

ALGORITHM *Binomial*(n, k)

```
//Computes C(n, k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers n ≥ k ≥ 0
//Output: The value of C(n, k)
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i
            C[i, j] ← 1
        else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, k]
```

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Thank You