|  |
| --- |
| Swayam |
| **Introduction to Algorithms and Analysis** |
| Prof. Sourav Mukhopadhyay | IIT Kharagpur |

|  |
| --- |
| VARUN KUMAR  02-11-2025 |

Table of Contents

[Week – 01: Sorting Problem, Time Complexity, and Asymptotic Analysis 3](#_Toc201089786)

[Lecture 01: Insertion Sort 3](#_Toc201089787)

[Lecture 02: Analysis of Insertion Sort 4](#_Toc201089788)

[Lecture 03: Asymptotic Notation 5](#_Toc201089789)

[Lecture 04: Recurrence for Merge Sort 6](#_Toc201089790)

[Lecture 05: Substitution Method 8](#_Toc201089791)

[Week – 02: Solving Recurrence, Divide and Conquer 10](#_Toc201089792)

[Lecture 06: The Master Method 10](#_Toc201089793)

[Lecture 07: Divide & Conquer 14](#_Toc201089794)

[Lecture 08: Divide & Conquer [Contd….] 15](#_Toc201089795)

[Lecture 09: Strassen’s Algorithm 16](#_Toc201089796)

[Lecture 10: Quick Sort 22](#_Toc201089797)

[Week – 03: Quick Sort and Heap Sort, Decision Tree 23](#_Toc201089798)

[Lecture 11 : Analysis of Quick Sort 23](#_Toc201089799)

[Week – 04: Linear Time Sorting, Order Statistic 24](#_Toc201089800)

[Week – 05: Hash Function, Binary Search Tree (BST) Sort 25](#_Toc201089801)

[Week – 06: Randomly Build BST, Red Black Tree, Augmentation of Data Structure 26](#_Toc201089802)

[Week – 07: Van Emde Boas, Amortized Analysis, Computational Geometry 27](#_Toc201089803)

[Week – 08: Dynamic Programming, Graph, Prim’s Algorithm 28](#_Toc201089804)

[Week – 09: BFS & DFS, Shortest Path Problem, Dijktra, Bellman-Ford 29](#_Toc201089805)

[Week – 10: All Pair Shortest Path, Floyd-Warshall, Jhonson Algorithm 30](#_Toc201089806)

[Week – 11: More Amortized Analysis, Disjoint Set Data Structure 31](#_Toc201089807)

[Week – 12: Network Flow, Computational Complexity 32](#_Toc201089808)

[Appendix – 01: Test 33](#_Toc201089809)

[Week - 01 33](#_Toc201089810)

[2023 33](#_Toc201089811)

[2025 35](#_Toc201089812)

[Week – 02 36](#_Toc201089813)

[2023 36](#_Toc201089814)

[2025 38](#_Toc201089815)

[Appendix – 02: Important Links 39](#_Toc201089816)

[Appendix – 03: Chat GPT and Deep Seek 40](#_Toc201089817)

[Insertion Sort 40](#_Toc201089818)

[Whimsical Diagrams 44](#_Toc201089819)

[Sorting Technique 44](#_Toc201089820)

[Divide and Conquer Algorithm 45](#_Toc201089821)

[Greedy Algorithm 46](#_Toc201089822)

[Dynamic Programming 47](#_Toc201089823)

[Master’s Theorem 48](#_Toc201089824)

[Appendix – 04: Python Setup Guide 49](#_Toc201089825)

[Appendix – 05: Step-by-Step Guide of Various Algorithm with Python Code 52](#_Toc201089826)

[01 – Implementation of Dijkstra’s Algorithm 52](#_Toc201089827)

[02 - Implementation of Bellman-Ford Algorithm 53](#_Toc201089828)

[03 - Implementation of Kahn’s Algorithm 54](#_Toc201089829)

[04 - Implementation of Dinic’s Algorithm 55](#_Toc201089830)

[05 - Implementation of Ford-Fulkerson Algorithm 56](#_Toc201089831)

[06 - Implementation of Prim’s Algorithm 57](#_Toc201089832)

[07 - Implementation of Kruskal’s Algorithm 58](#_Toc201089833)

[08 - Implementation of Basic Operation Associated with B+ Tree 59](#_Toc201089834)

[09 - Implementation of K – Dimensional Tree 60](#_Toc201089835)

[10 - Implementation of Rabin-Krap Algorithm 61](#_Toc201089836)

[11 - Implementation of KMP Algorithm 64](#_Toc201089837)

[12 - Implementation of Union by Rank Algorithm 65](#_Toc201089838)

[13 - Implementation of Various Sorting Algorithm 66](#_Toc201089839)

[14- Implementation of Quick Sort Algorithm 67](#_Toc201089840)

[15- Implementation of Merge Sort Algorithm 74](#_Toc201089841)

[16 - Implementation of Heap Sort Algorithm 79](#_Toc201089842)

[Appendix – 06: Working with Graph using NetworkX 89](#_Toc201089843)

[Appendix – 07: Essential Problems from CLRS 94](#_Toc201089844)

# Week – 01: Sorting Problem, Time Complexity, and Asymptotic Analysis

## Lecture 01: Insertion Sort

**Topics to be Covered: -**

* Problem of Sorting, Pseudo Code,
* Insertion Sort, Loop Invariant, Runtime, Parameterise the runtime by the size of the input

Ink Drawings
Input:- A sequence of < a1, a2, ...., an> of numbers
Output:- A permutation of < a1', a2', ...., an'> such that

a1' ≤ a2' ≤ .... ≤ an
The Problem of Sorting


Ink Drawings
Example
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Pseudo Code:- Insertion Sort
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


**Do a Dry Run of the code**.

Ink Drawings
Example of Insertion Sort
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Runtime of Insertion Sort
Runtime of Insertion Sort

The running time depends on input i.e., already sorted sequence is easier to sort
Parameterize the running time by the size of the input, since short sequence are easier to sort than long one.
Generally, we seek upper bounds on running time.


## Lecture 02: Analysis of Insertion Sort

**Topics to be Covered: -**

* Types of Analysis: Worst Case, Best Case and Average Case, Machine Independency
* Asymptotic Notation, Big-Theta Notation ()

Ink Drawings
Types of Analysis
Types of Analysis

Worst Case (Usually) :- 
T(n) = Maximum time of algorithm on any input of size 'n'.

Average Case (Sometimes) :- 
T(n) = Expected time of algorithm on any input of size 'n'.

Best Case :- 
Cheat with a slow algorithm that works fast on 'some' input.


Ink Drawings
Machine-Independent Time
Machine-Independent Time
What is Insertion Sort's worst-case time?
It depends on the speed of the computer
Relative Speed (on the same machine)
Absolute Speed (on different machine)
Big Idea
Big Idea
Ignore machine-dependent constants.
Look at 'growth' of T(n) as n →∞

ASYMPTOTIC ANALYSIS


Ink Drawings
𝛉 Notation
𝛉 Notation

Maths:-
𝜽﷐𝒈﷐𝒏﷯﷯ = {𝑓﷐𝑛﷯ : ∃ 𝑝𝑜𝑠𝑖𝑡𝑖𝑣𝑒 𝑐𝑜𝑛𝑠𝑡𝑎𝑛𝑡 c1, c2 and n0 such that 0 ≤ c1 . g(n) ≤ f(n) ≤ c2 . g(n), ∀ n ≥ n0}

Engineering:- 
Drop low order terms; ignore leading constants

Example:- 
3n - 90n +5n - 1024 = θ﷐﷐𝑛﷮3﷯﷯
Ink Drawings


## Lecture 03: Asymptotic Notation

**Topics to be Covered: -**

* Asymptotic Notation: - Big-Oh, Big-Theta, and Big-Omega
* Time Complexity of Insertion Sort: - Worst Case, Best Case, and Average Case
* Merge Sort

Ink Drawings
𝚶 Notation 

Ο﷐𝑔﷐𝑛﷯﷯ = { f(n) : ∃ positive constant c and n0 such that f(n) ≤ c * g(n) ∀ n ≥ n0}
𝚶 Notation 

Ο﷐𝑔﷐𝑛﷯﷯ = { f(n) : ∃ positive constant c and n0 such that f(n) ≤ c * g(n) ∀ n ≥ n0}
 𝛀 Notation 

Ω﷐𝑔﷐𝑛﷯﷯ = { f(n) : ∃ positive constant c and n0 such that f(n) ≥ c * g(n) ∀ n ≥ n0}
 𝛀 Notation 

Ω﷐𝑔﷐𝑛﷯﷯ = { f(n) : ∃ positive constant c and n0 such that f(n) ≥ c * g(n) ∀ n ≥ n0}


Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Asymptotic Notation
Asymptotic Notation
When n gets large enough a 𝜃﷐﷐𝑛﷮2﷯﷯ algorithm always beats a Ο﷐﷐𝑛﷮3﷯﷯ algorithm.
We shouldn't ignore asymptotically slower algorithm.
Real world design situations often calls for a careful balancing of engineering objectives.
It is a useful tool to help structure our thinking.
Insertion Sort Analysis
Insertion Sort Analysis

Worst Case: Input Inversely sorted.

T(n) = ﷐𝑗=2﷮𝑛﷮𝜃﷐𝑗﷯﷯= 𝜃﷐﷐𝑛﷮2﷯﷯ [Arithmetic Series]

Average Case: All permutation equally likely.

T(n) = ﷐𝑗=2﷮𝑛﷮𝜃﷐﷐𝑗﷮2﷯﷯﷯= 𝜃﷐﷐𝑛﷮2﷯﷯

It is moderately fast for small 'n'.
It is not at all fast for large 'n'.
Ink Drawings
Ink Drawings


## Lecture 04: Recurrence for Merge Sort

Topics to be Covered: -

* Merge Sort, Run time of Merge Sort
* Recurrence and Recursive Tree

Ink Drawings
Merge Sort
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Analysis of Merge Sort
Analysis of Merge Sort
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Recurrence for Merge Sort
Recurrence for Merge Sort
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
We shall usually omit the base case when  for sufficiently small 'n' and when it has no effect on the solution to the recurrence


Ink Drawings
Recursion Tree
Recursion Tree
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Best case of Merge Sort
Best case of Merge Sort
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


## Lecture 05: Substitution Method

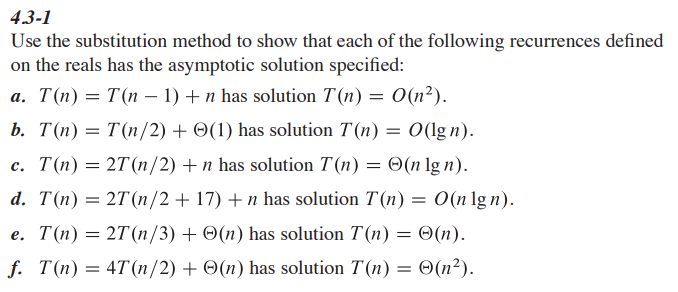
Topics to be Covered: -

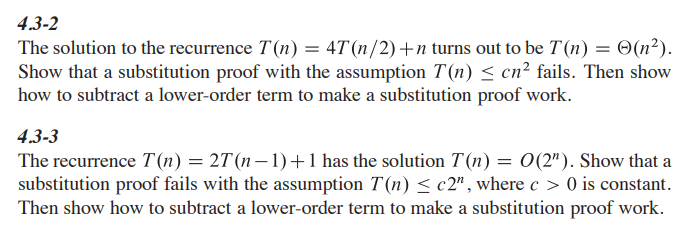
* Solving the Recurrence: Substitution Method
* Method of Induction

It is the most general method:

* Guess the form of solution
* Verify by Induction
* Solve for constants

Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings

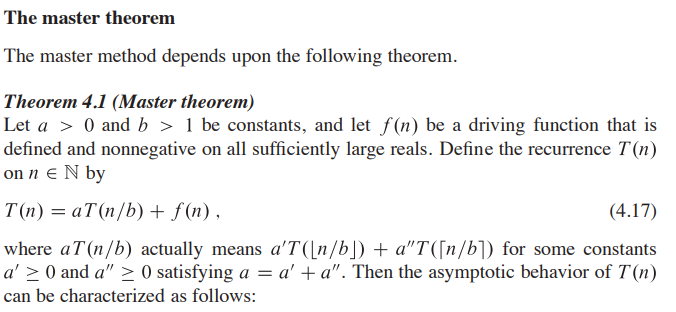


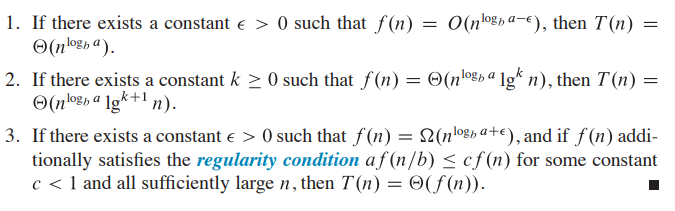
# Week – 02: Solving Recurrence, Divide and Conquer

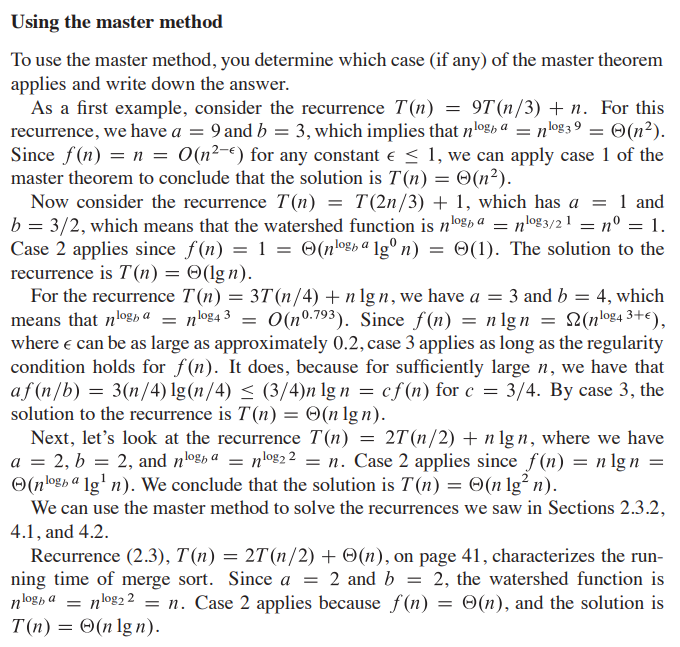
## Lecture 06: The Master Method

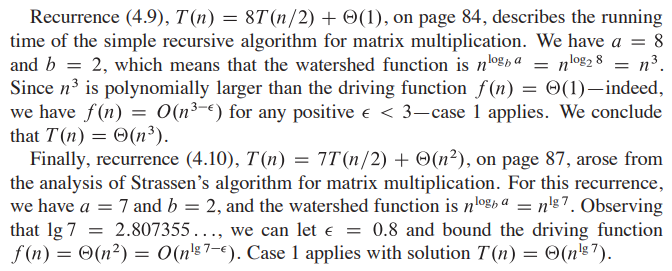
Topics to be Covered: -

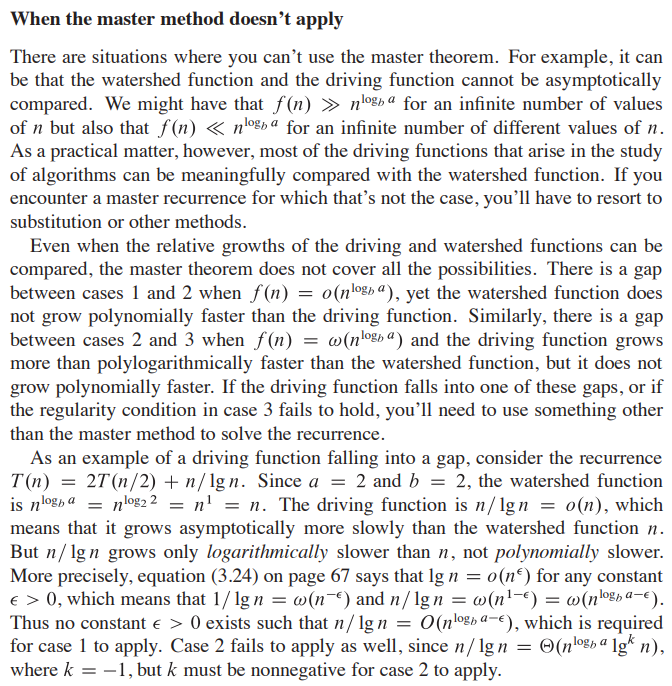
* Solving the recurrence of the form T(n) = aT(n/b) + f(n),
* Master method

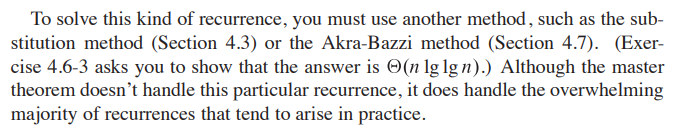
[[1]](#footnote-1)

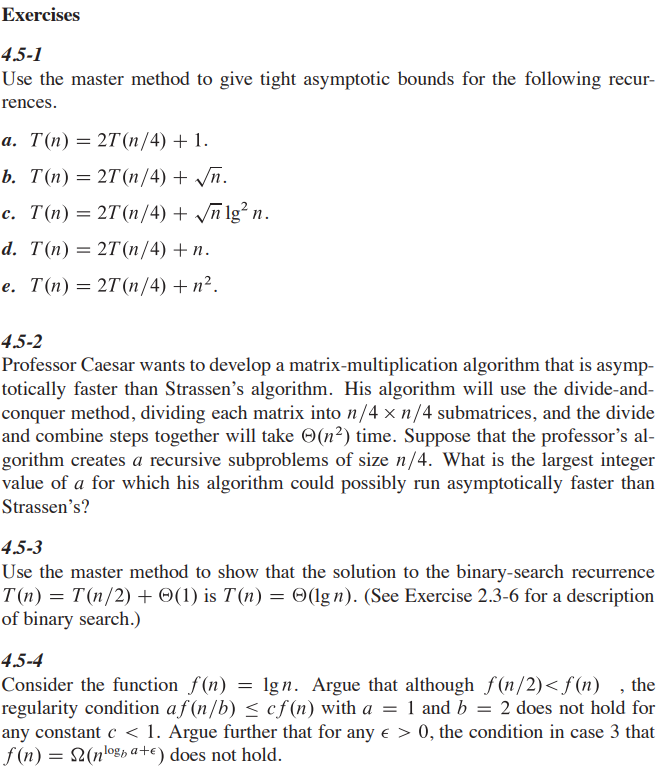


[[2]](#footnote-2)



[[3]](#footnote-3)



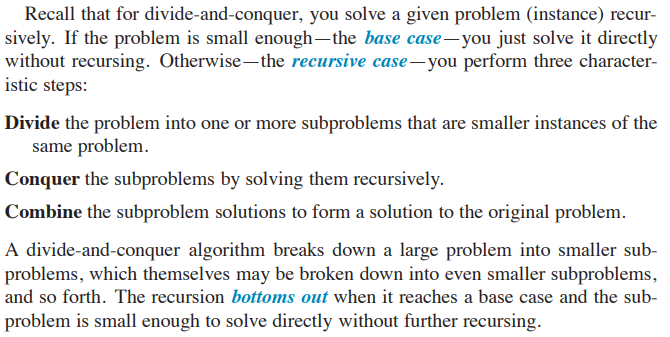
[[4]](#footnote-4)

**Read section 4.6 for better understanding.**

## Lecture 07: Divide & Conquer

Topics to be Covered:

* Divide and Conquer: Design Paradigm
* Binary Search
* Powering a Number

[[5]](#footnote-5)

Ink Drawings
Binary Search
Binary Search
Divide :- Check Middle Element
Conquer :- Recursively search one sub array
Combine :- Trivial (Not needed in case of Binary Search)
Recurrence for Binary Search
𝑻﷐𝒏﷯=𝟏 ∗𝑻﷐﷐𝒏﷮𝟐﷯﷯+ 𝜽﷐𝒏﷯
﷐𝑛﷮﷐﷐log﷮𝑏﷯﷮𝑎﷯﷯ = ﷐𝑛﷮﷐﷐log﷮2﷯﷮1﷯﷯ = ﷐𝑛﷮0﷯ =  1 ⇒ CASE II ( k = 0 )
𝑻﷐𝒏﷯= 𝜽﷐﷐𝐥𝐨𝐠﷮𝒏﷯﷯
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Powering a Number
Powering a Number
Problem:- Compute an, n ∈ N
Naive Algorithm:- 𝜃﷐𝑛﷯
Divide and Conquer Algorithm:

Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


## Lecture 08: Divide & Conquer [Contd….]

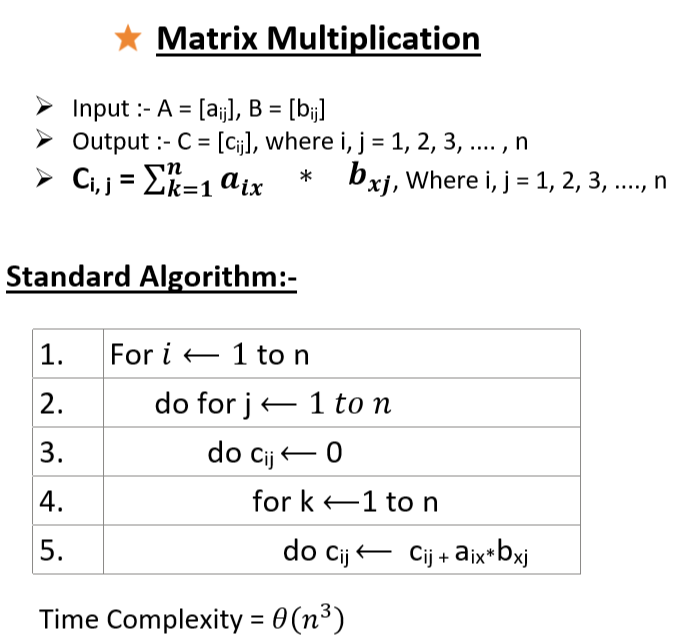
Topics to be Covered:

* Fibonacci Number
* Strassen’s Matrix Multiplication Algorithm

Ink Drawings
Fibonacci Number
Fibonacci Number
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings

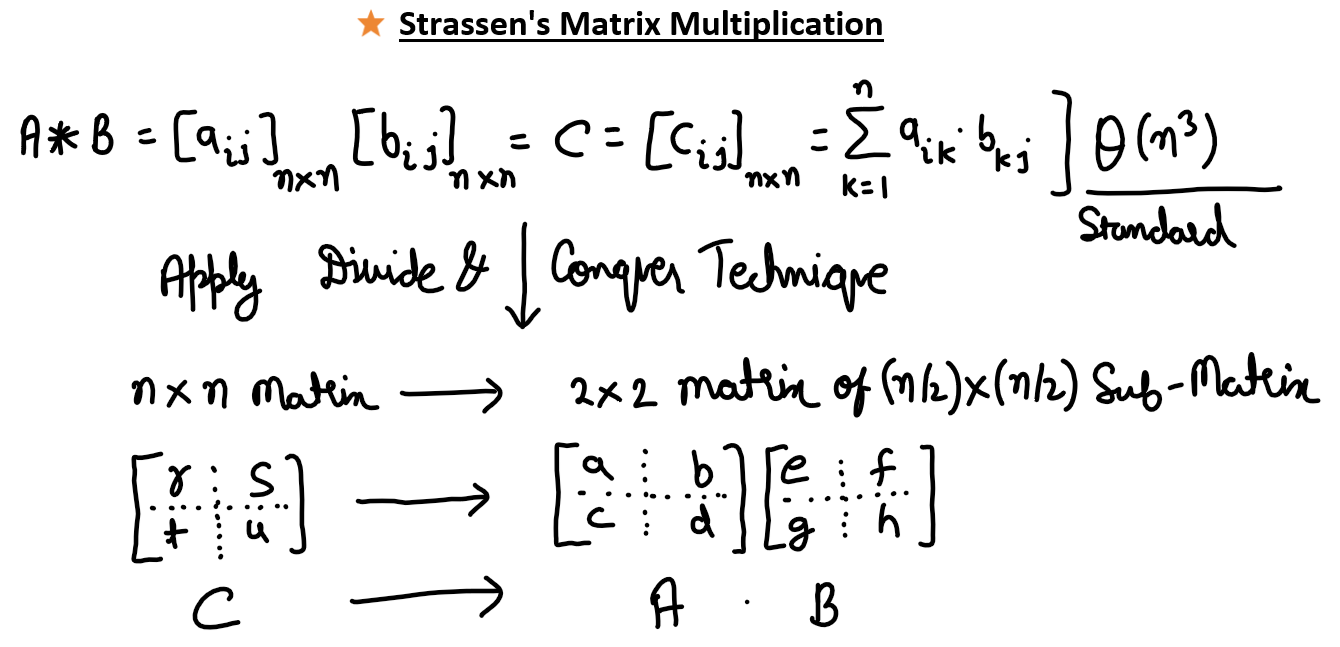

Ink Drawings
Recursive Squaring
Recursive Squaring
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings

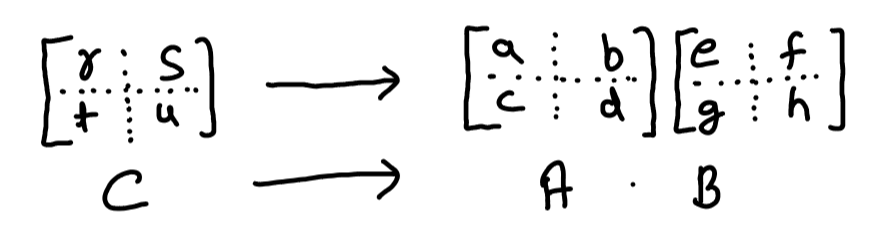



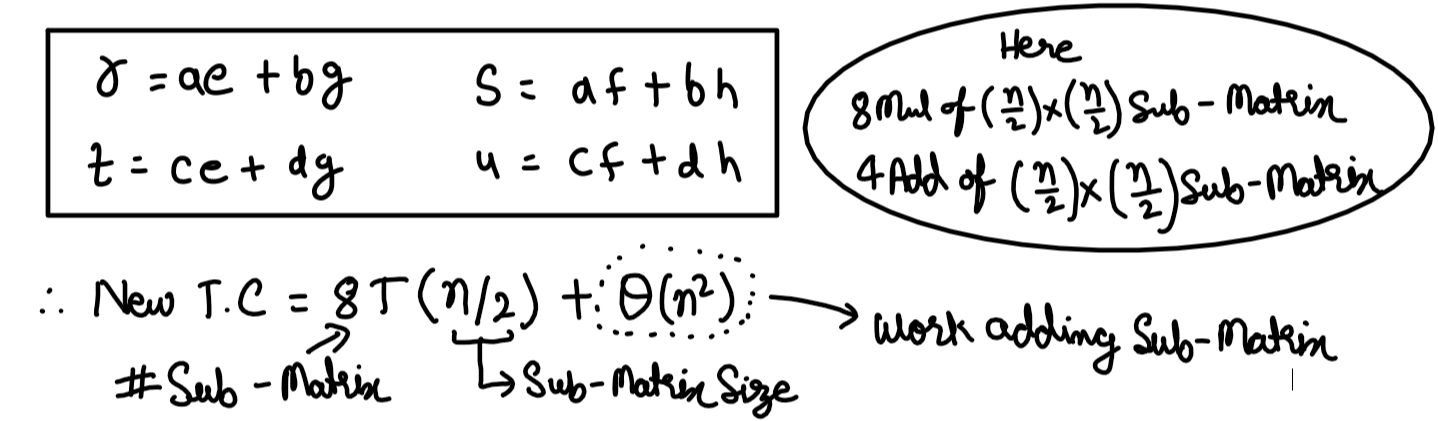
## Lecture 09: Strassen’s Algorithm

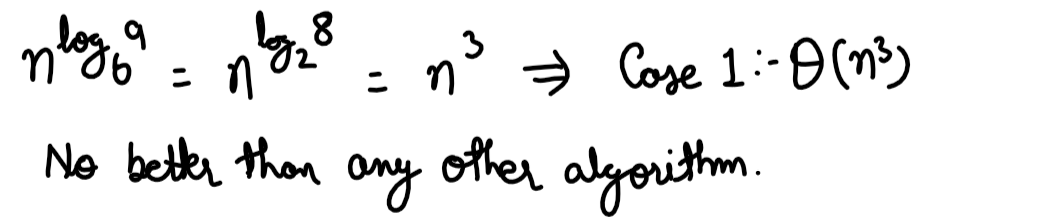
Topics to be Covered:

* Matrix Multiplication using Divide & Conquer
* Strassen’s Idea









**✅ Strassen’s Matrix Multiplication Algorithm [[6]](#footnote-6)— Explained Simply**

Strassen’s algorithm is a **faster matrix multiplication** method than the standard algorithm. It reduces the number of **multiplications** required by using a **divide-and-conquer** technique.

**🧠 Idea Behind Strassen’s Algorithm**

For two square matrices of size:

* Normally, to multiply, we do n3 scalar multiplications.
* Strassen reduces this to approximately by doing **7 multiplications** instead of 8 when dividing matrices.

It works **only when n is a power of 2**, or we **pad** the matrix to make it so.

**🧩 Step-by-Step in Plain English**

Assume two matrices **A** and **B**, both , where n is a power of 2.

**Step 1: Divide matrices into 4 submatrices**

Divide A and B like this:

|  |
| --- |
| A = | A11 A12 | B = | B11 B12 |  | A21 A22 | | B21 B22 | |

Each block is of size

**Step 2: Compute 7 products (instead of 8)**

We define 7 matrix products:

|  |
| --- |
| M1 = (A11 + A22) × (B11 + B22)  M2 = (A21 + A22) × B11  M3 = A11 × (B12 - B22)  M4 = A22 × (B21 - B11)  M5 = (A11 + A12) × B22  M6 = (A21 - A11) × (B11 + B12)  M7 = (A12 - A22) × (B21 + B22) |

These involve **7 recursive multiplications** of size

**Step 3: Combine to get result matrix C**

We compute the submatrices of result matrix C:

|  |
| --- |
| C11 = M1 + M4 - M5 + M7  C12 = M3 + M5  C21 = M2 + M4  C22 = M1 - M2 + M3 + M6 |

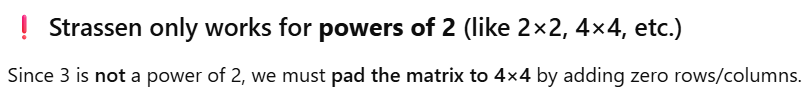
Then combine into full matrix C.

**🔍 Explanation of Efficiency**

* **Standard method**: multiplications
* **Strassen**: Reduces multiplication count from 8 to 7
* **Time Complexity**:
  + Solving this gives:

**📝 Final Notes**

* Better for **large matrices**.
* **Numerical instability** can be an issue for floating-point arithmetic.
* Padding may be needed if size isn't a power of 2.



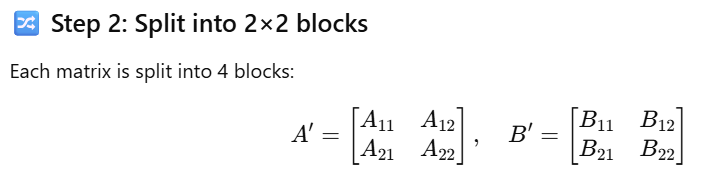
**🧮 Let’s define two 3×3 matrices:**

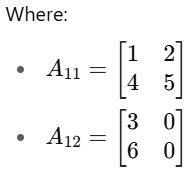
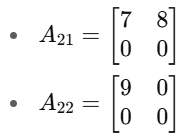
Let

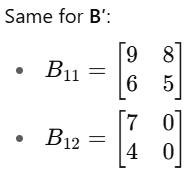
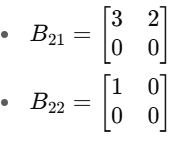
and

🔄 Step 1: Pad to 4×4 matrices

and

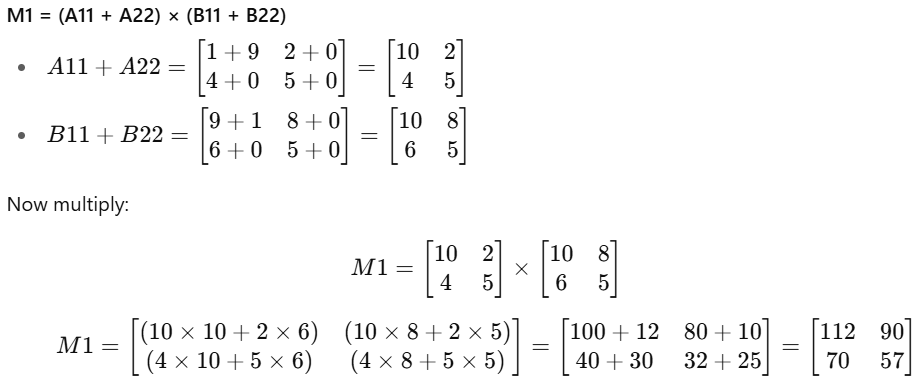


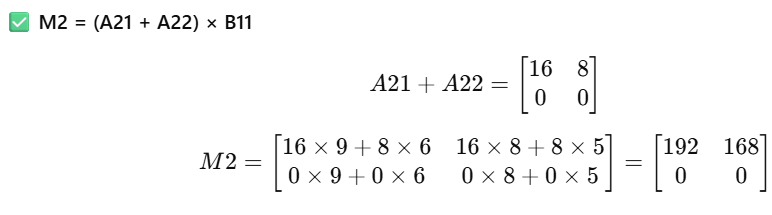
 

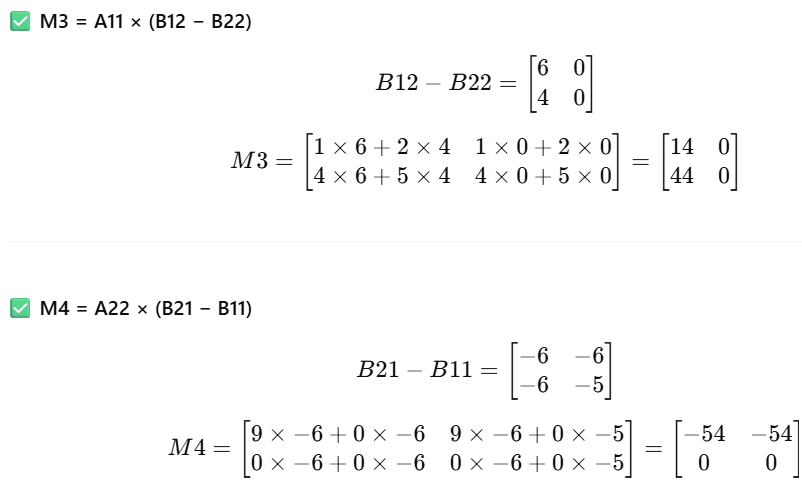
 

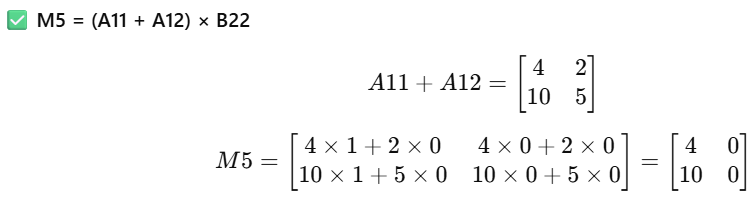
**🧠 Step 3: Compute the 7 Strassen products (M1–M7)**

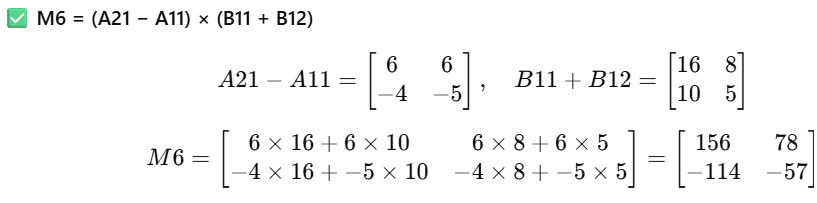
Let’s calculate just **M1** in detail to illustrate:

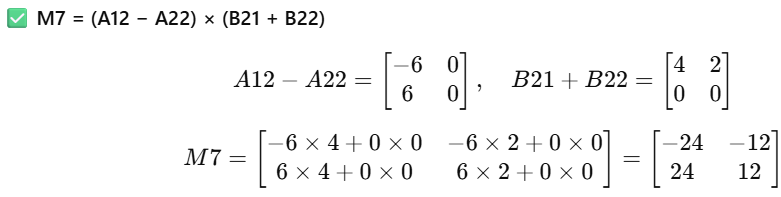


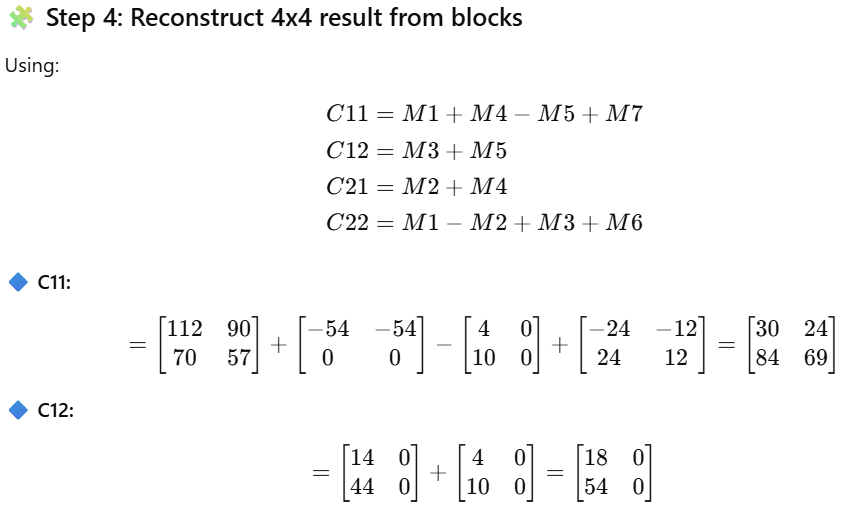


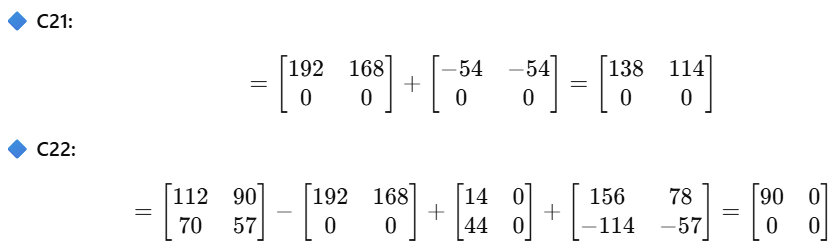


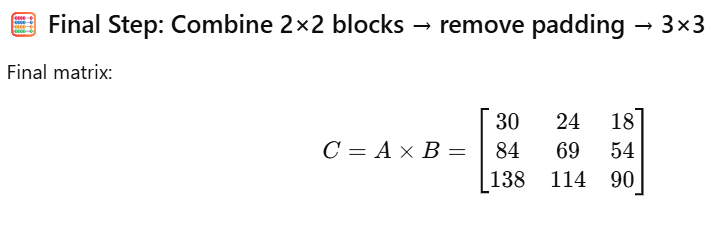












## Lecture 10: Quick Sort

Topics to be Covered:

* Pseudo Code of Quick Sort
* Worst Case, Best Case and Almost Best Case
* Good Pivot and Bad Pivot

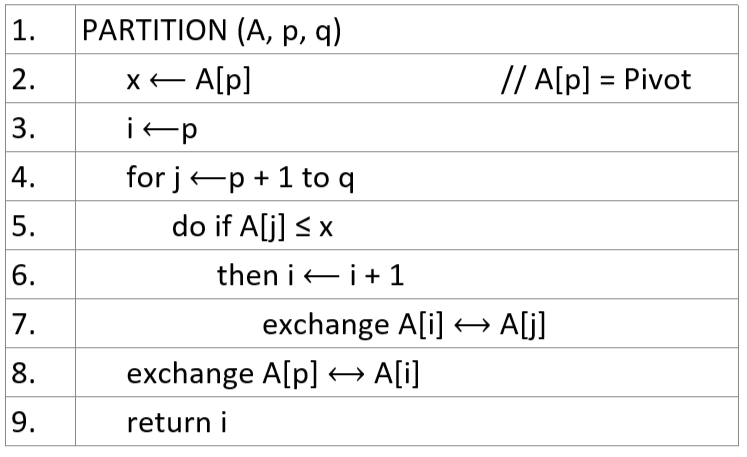
1. **Divide**: - Partition the array into two sub-arrays around a pivot x such that elements in lower sub-array ≤ x ≤ elements in upper sub-array



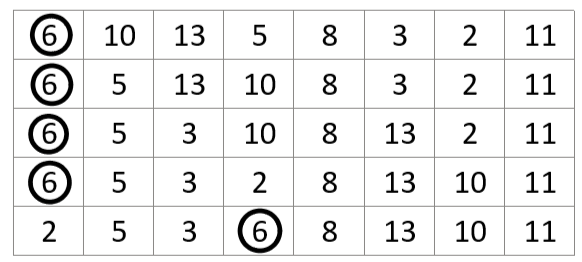
1. **Conque**r: - Recursively sort 2 sub-arrays
2. **Combine**: - Trivial

**Key**: - Linear Time Partition Sub-Routine

**Partitioning Subroutine**



**Example of Partitioning**



# Week – 03: Quick Sort and Heap Sort, Decision Tree

## Lecture 11 : Analysis of Quick Sort

# Week – 04: Linear Time Sorting, Order Statistic

# Week – 05: Hash Function, Binary Search Tree (BST) Sort

# Week – 06: Randomly Build BST, Red Black Tree, Augmentation of Data Structure

# Week – 07: Van Emde Boas, Amortized Analysis, Computational Geometry

# Week – 08: Dynamic Programming, Graph, Prim’s Algorithm

# Week – 09: BFS & DFS, Shortest Path Problem, Dijktra, Bellman-Ford

# Week – 10: All Pair Shortest Path, Floyd-Warshall, Jhonson Algorithm

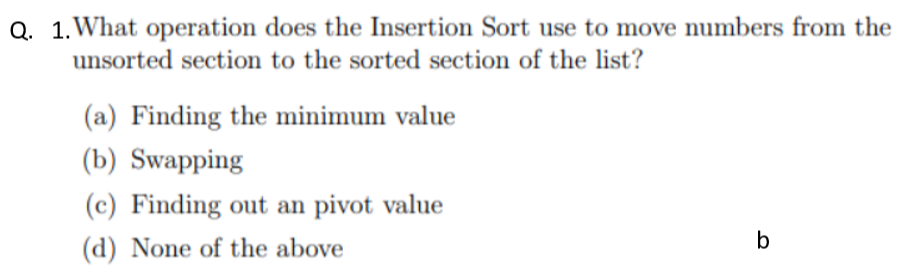
# Week – 11: More Amortized Analysis, Disjoint Set Data Structure

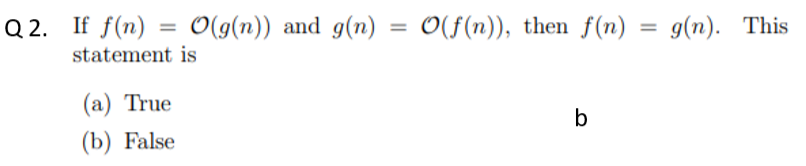
# Week – 12: Network Flow, Computational Complexity

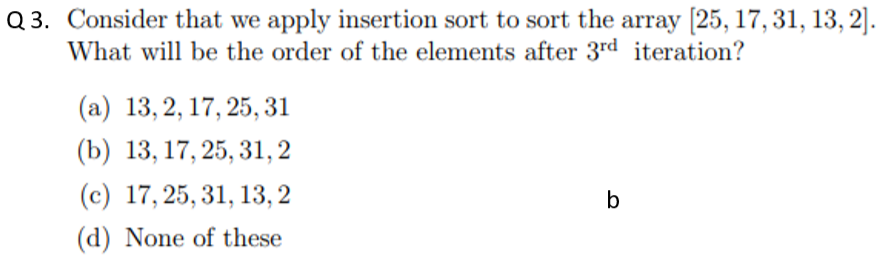
# Appendix – 01: Test

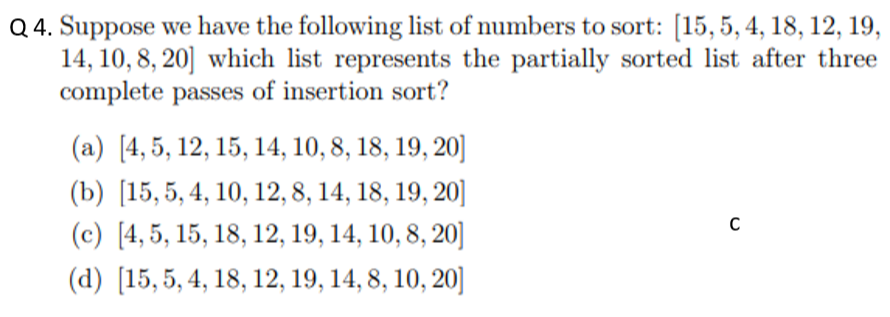
## Week - 01

### 2023

[[7]](#footnote-7)







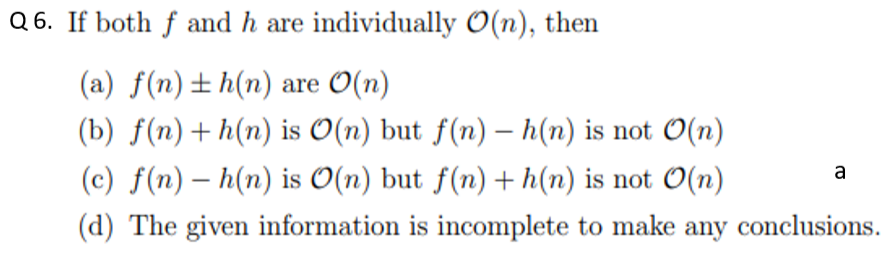
Q 5. is

(a)

(b)

(c)

(d) (c) is the correct answer



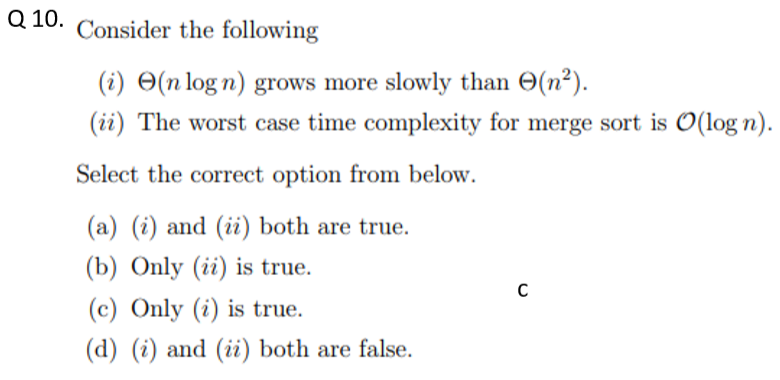
**Q 7. The space complexity of Merge sort is**

1. (c) is the correct answer

**Q 8. Merge sort uses**

1. Divide and Conquer strategy
2. Backtracking Strategy
3. Heuristic Search
4. Greedy Approach (a) is the correct answer

**Q 9. If all the element of the given array is equal for example {1, 1, 1, 1, 1, 1}, What woud be running time of Insertion Sort?**



### 2025

## Week – 02

### 2023

Q 1. To …………………, Master’s Theorem is used.

1. Solving Recurrence
2. Solving Iterative relation
3. Analysing loops
4. None of the above (a) is the correct answer

Q 2. Time complexity of recursive matrix multiplication using Divide and Conquer Method is ……………….?

1. (c) is the correct answer

Q 3. Using median-of-three partition method, what is the pivot element of the following strings?

[8, 1, 4, 9, 6, 3, 5, 2, 7, 0]

1. 8
2. 7
3. 6
4. 9 (c) is the correct answer

Q 4. In Strassen’s Matrix Multiplication, what is the formula to calculate the element present in second row, first column of the product matrix?

(Here the notation is defined as in the lecture notes)

1. P1 + P7
2. P1 + P3
3. P2 + P4 – P5 + P7
4. P3 + P4  (d) is the correct answer

Q 5. Solve the following recurrence using Master’s Theorem

1. (a) is the correct answer

Q 6. Multiplying a matrix by an matrix, using Strassen’s algorithm as a sub routine produce

1. k multiplication of matrices
2. multiplication of matrices
3. multiplication of matrices
4. multiplication of matrices (b) is the correct answer

Q 7. Given the following list of number [14, 17, 13, 15, 19, 10, 3, 16, 9, 12] which answer shows the content of the list after the second partitioning according to Quick Sort algorithm?

1. [9, 3, 10, 13, 12]
2. [9, 3, 10, 13, 12, 14]
3. [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
4. [9, 3, 10, 13, 12, 14, 19, 16, 15, 17] (d) is the correct answer

Q 8. Can Master’s method be used to solve

1. Yes
2. No (b) is the correct answer

Q 9. The running time of Strassen’s Algorithm for Matrix Multiplication is

1. (a) is the correct answer

Q 10. Let **rn** be the number of n-bit strings that do NOT contain two consecutive 1’s. Which of the following is the recurrence relation for **rn**?

1. **rn** = **rn-1** + 2 **rn-2**
2. **rn** = **rn-1** + **rn-2**
3. **rn** = 2**rn-1** + **rn-2**
4. **rn** = **rn-1** + 2 **rn-2** (b) is the correct answer

### 2025

# Appendix – 02: Important Links

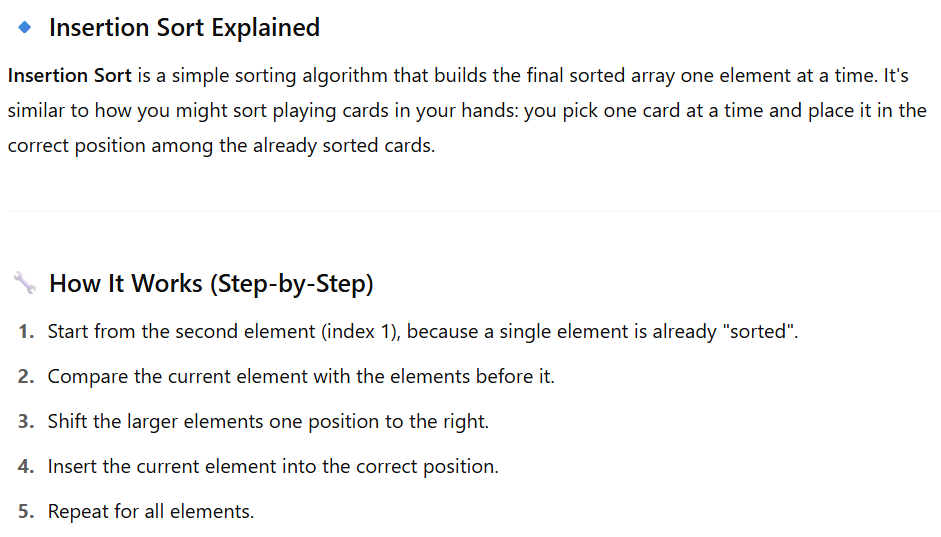
Current: - <https://onlinecourses.nptel.ac.in/noc25_cs150/preview>

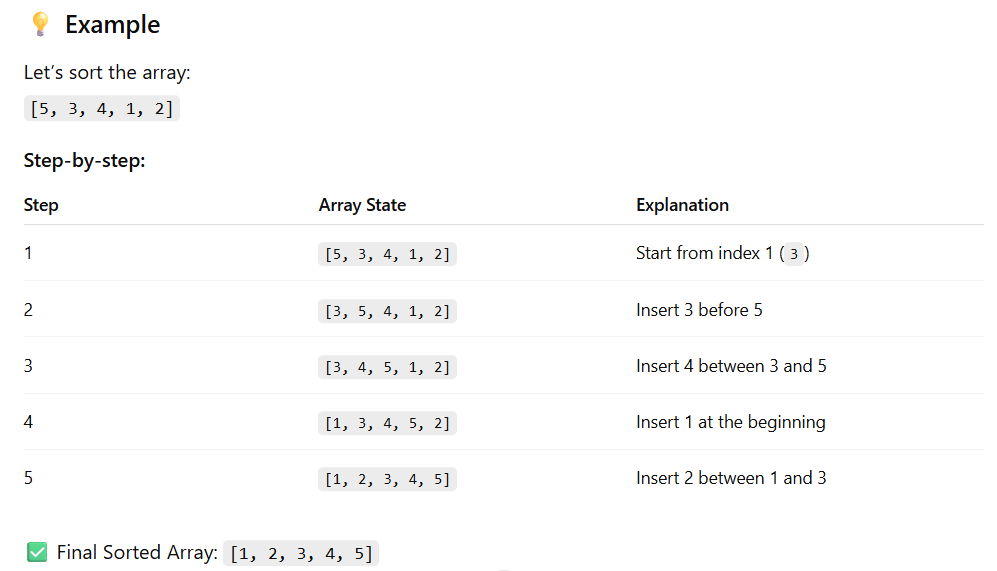
2023: - <https://onlinecourses.nptel.ac.in/noc23_cs88/preview>

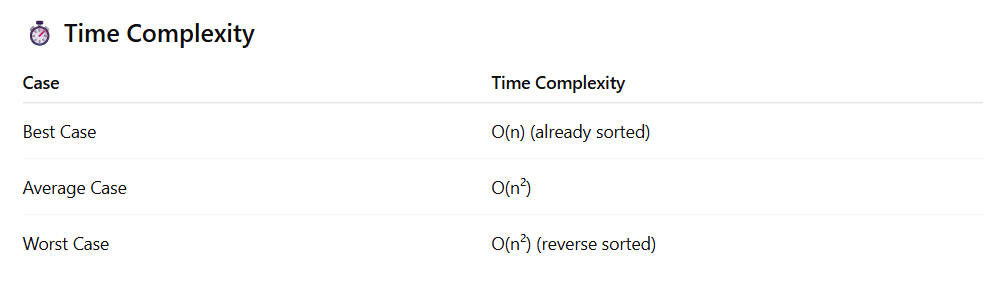
NPTEL: - <https://nptel.ac.in/courses/106105164>

# Appendix – 03: Chat GPT and Deep Seek

## Insertion Sort







|  |
| --- |
| # Python Code for Insertion Sort  def insertion\_sort(arr):  for i in range(1, len(arr)):  key = arr[i]  j = i - 1    # Move elements greater than key to one position ahead  while j >= 0 and arr[j] > key:  arr[j + 1] = arr[j]  j -= 1    arr[j + 1] = key  # Example  arr = [5, 3, 4, 1, 2]  insertion\_sort(arr)  print("Sorted array:", arr) |

Walk through the **step-by-step logic to code Insertion Sort**, assuming you understand loops, conditionals, and lists.

**🧩 Step-by-Step Plan to Code Insertion Sort**

Let’s say we want to sort the list arr = [5, 3, 4, 1, 2].

✅ **Step 1: Understand the Goal**

We want to sort arr in ascending order using the insertion sort technique.

**✅ Step 2: Loop Through the Array**

Start the loop from the second element (index 1), because the first element is trivially sorted.

**for i in range(1, len(arr)):**

**✅ Step 3: Store the Current Value**

Pick the current value (to be inserted into the sorted part on the left).

**key = arr[i]**

**✅ Step 4: Set Pointer for Comparison**

Set a pointer j to the element just before i. We'll use this to compare with the elements on the left side.

**j = i - 1**

**✅ Step 5: Shift Elements Greater Than key**

While j is not out of bounds and the current left-side element is greater than key, shift it to the right.

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j] # shift right

j -= 1 # move left

**✅ Step 6: Insert the key at the Correct Position**

Now insert the key at j + 1 because the last checked position (j) was either out of bounds or smaller than key.

arr[j + 1] = key

**✅ Step 7: Full Code Together**

See above

**✅ Step 8: Test It**

arr = [5, 3, 4, 1, 2]

insertion\_sort(arr)

print("Sorted array:", arr)

**Output:**

Sorted array: [1, 2, 3, 4, 5]

**🎯 Optional Tip for You**

Add print() statements to see what's happening at each step:

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

print(f"\nInserting {key}")

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

print(f"Moved {arr[j + 1]} to position {j + 2}")

arr[j + 1] = key

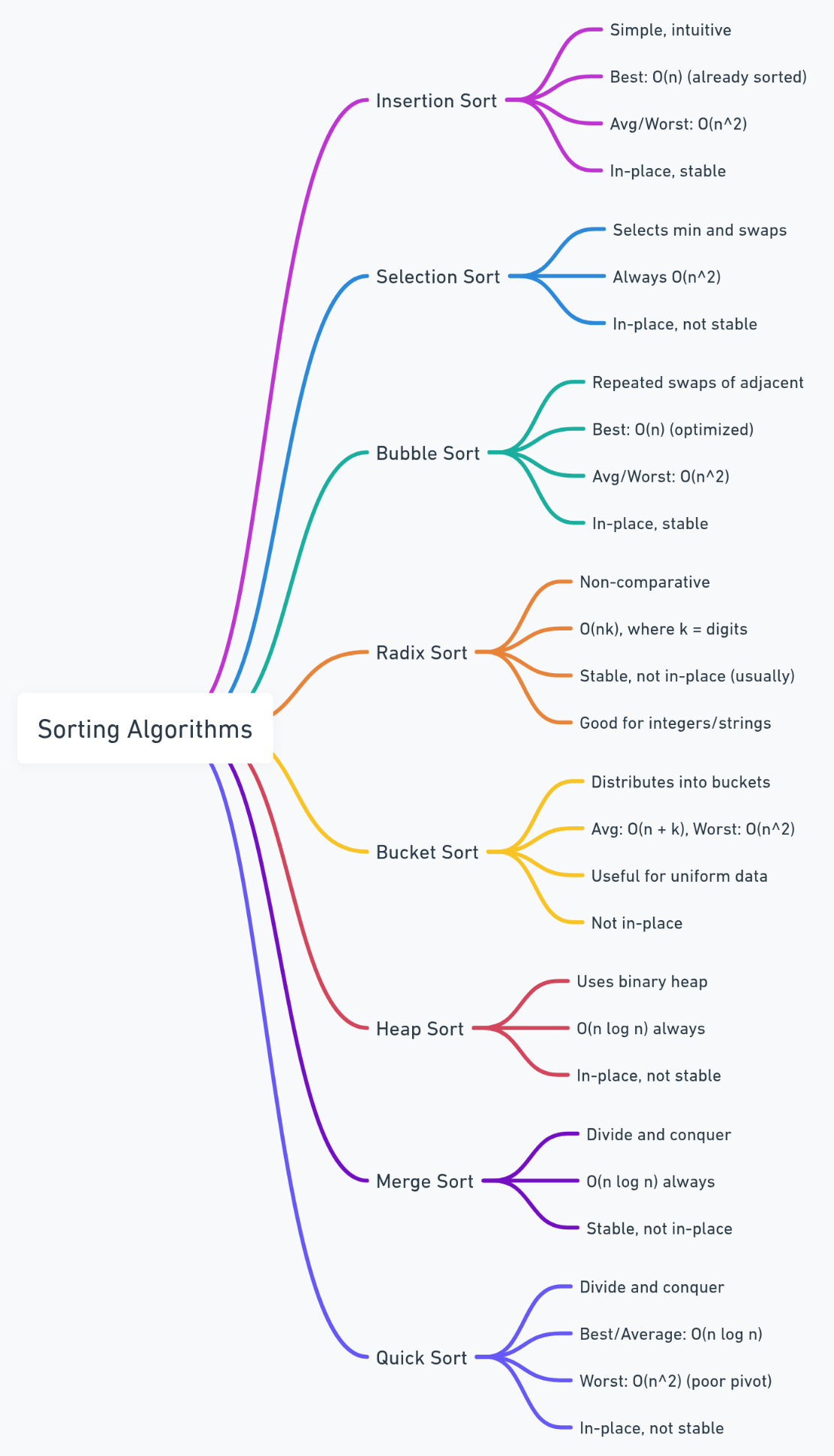
print(f"Inserted {key} at position {j + 1}")

print("Current array state:", arr)

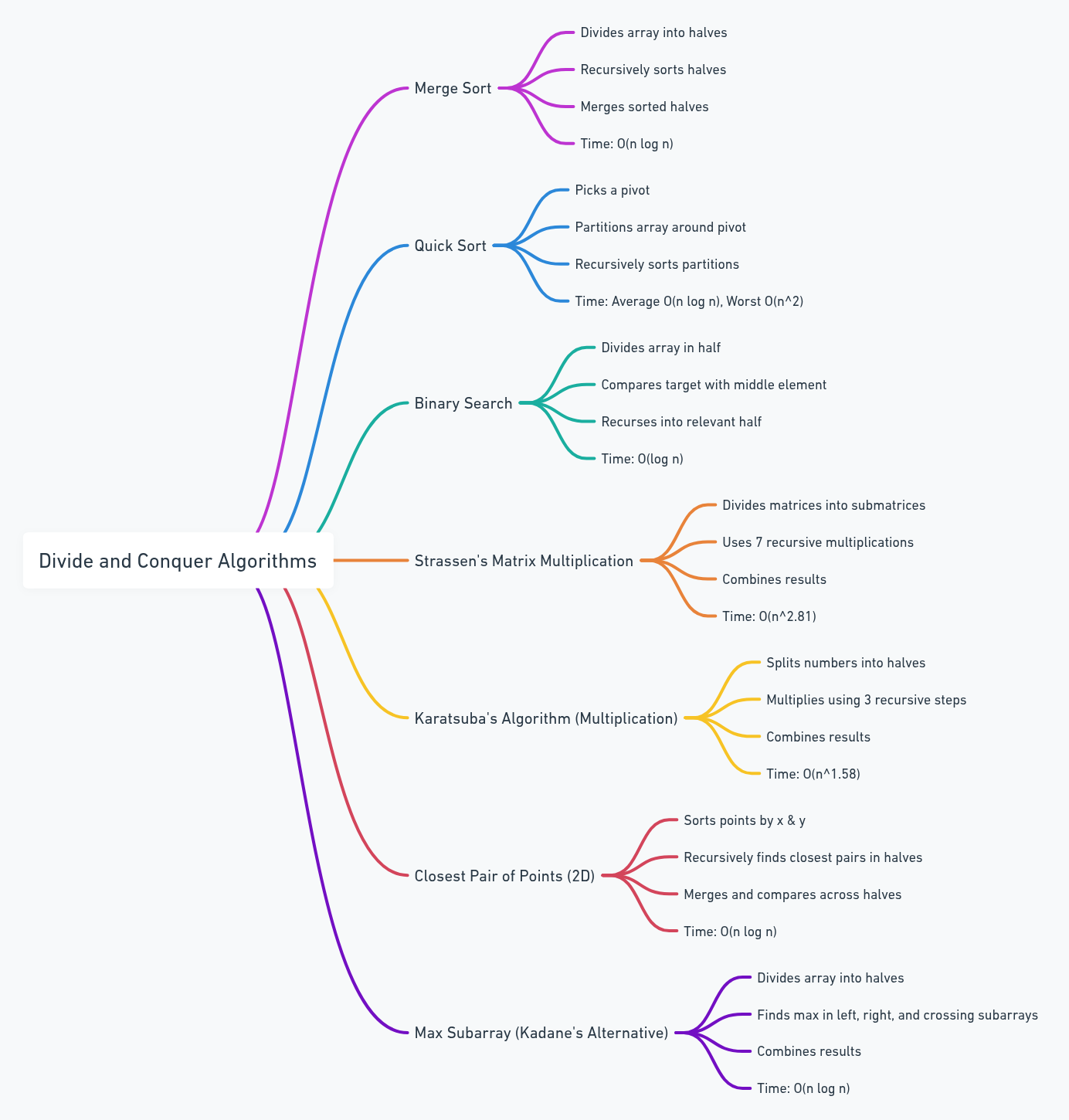
**Create Insertion Sort in Decreasing order.**

## Whimsical Diagrams

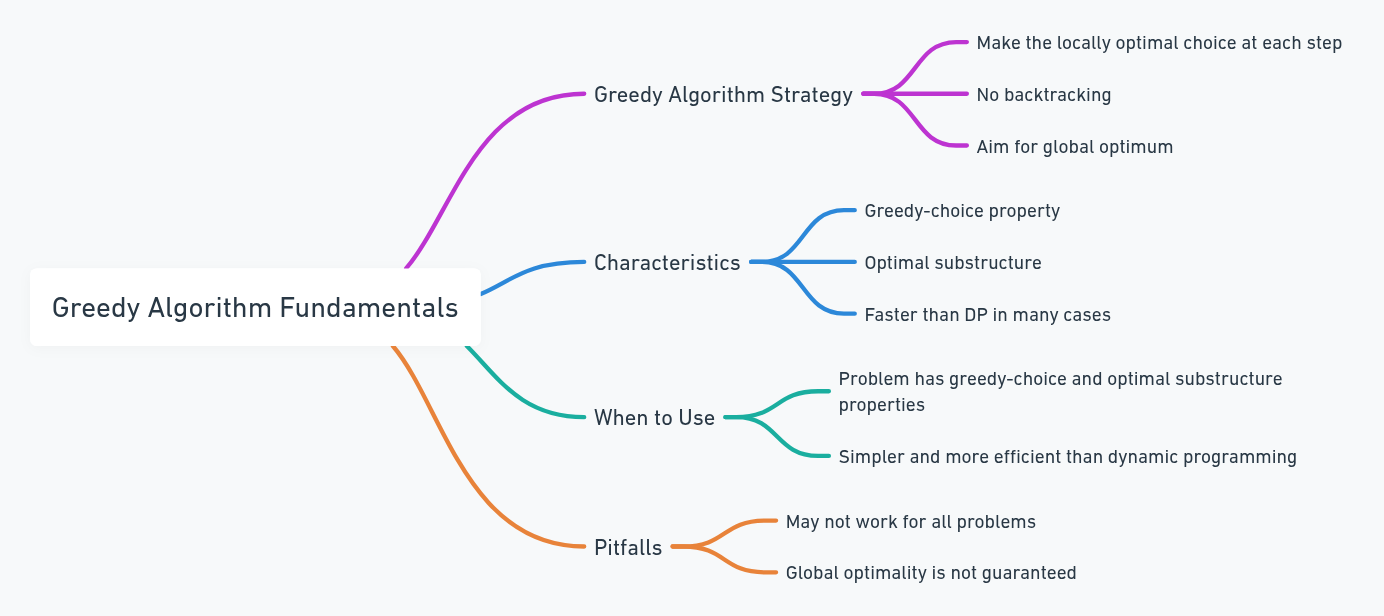
### Sorting Technique



### Divide and Conquer Algorithm

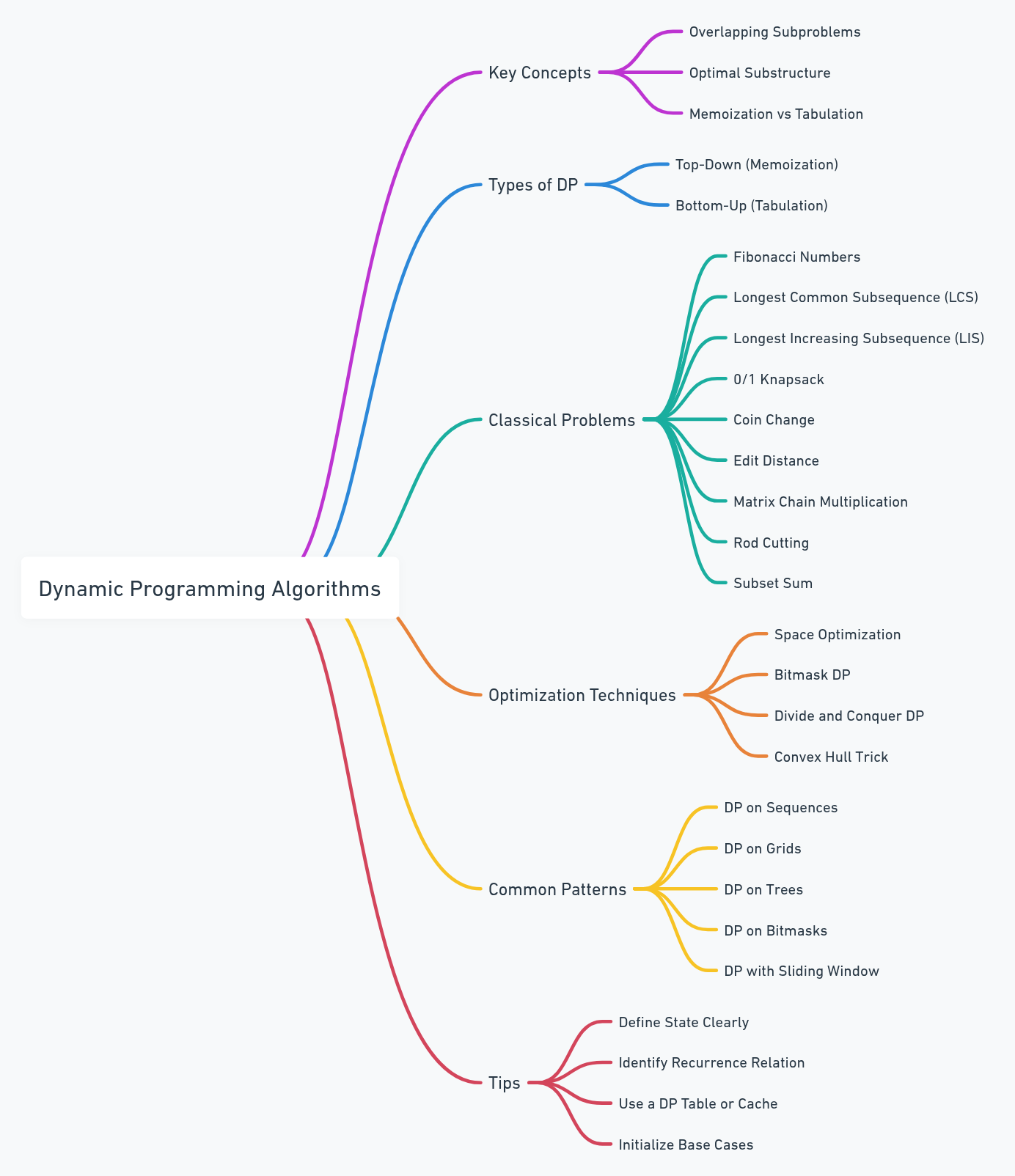


### Greedy Algorithm



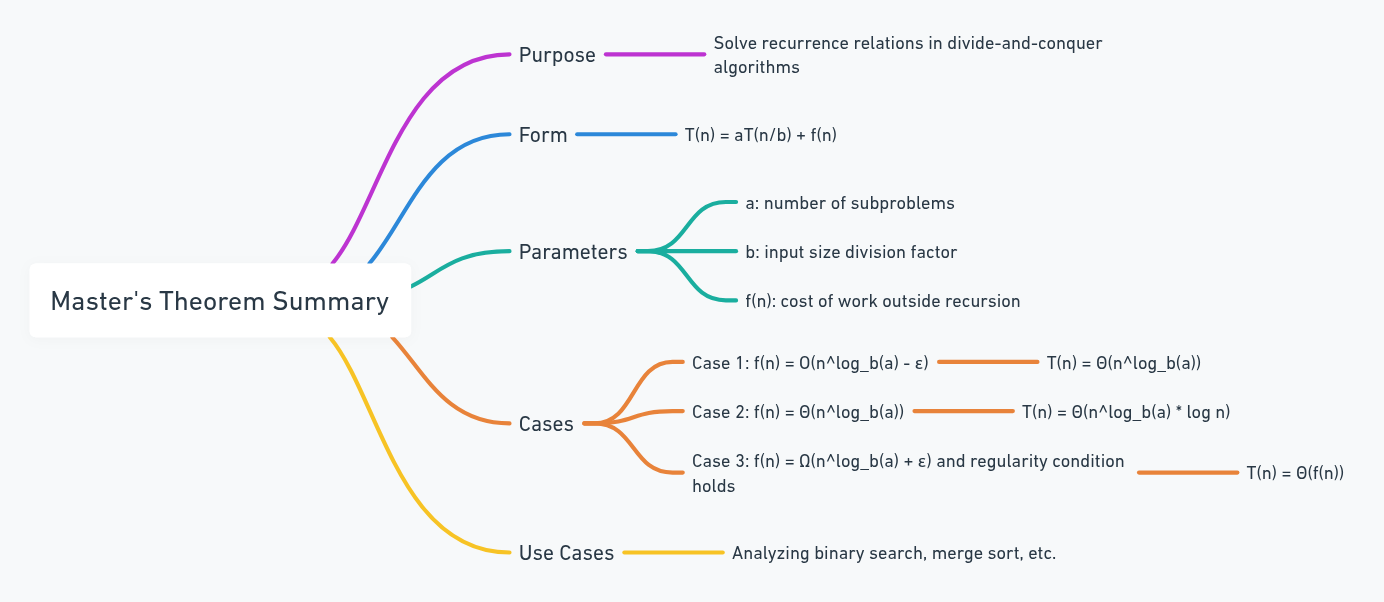


### Dynamic Programming





### Master’s Theorem



# Appendix – 04: Python Setup Guide

**🖥️ 1. Installing Python**

**🔹 Windows:**

1. Download the installer from: <https://www.python.org/downloads>
2. Run the installer:
   * ✅ Check **“Add Python to PATH”**
   * Click **Install Now**
3. Verify installation:

|  |
| --- |
| python –version |

**🔹 Linux (Ubuntu/Debian):**

|  |
| --- |
| sudo apt update  sudo apt install python3 python3-pip python3-venv  # Verify Installation  python –version |

**🧪 2. Create a Virtual Environment**

**🔹 Windows:**

|  |
| --- |
| python -m venv algo-env  .\algo-env\Scripts\activate |

**🔹 Linux:**

Bash

|  |
| --- |
| python3 -m venv algo-env  source algo-env/bin/activate |

**Deactivate with:**

|  |
| --- |
| deactivate |

**📦 3. Install Essential Packages**

Once the virtual environment is activated, install these recommended packages:

|  |
| --- |
| pip install jupyter matplotlib numpy pandas networkx rich |

🔍 Purpose of These Packages:

|  |  |
| --- | --- |
| **Package** | **Purpose** |
|  | Run notebooks for interactive coding |
|  | Visualization and plotting |
|  | Numerical computing |
|  | Data structures and manipulation |
|  | |  | | --- | |  |  |  | | --- | | Graph theory & algorithm practice | |
|  | |  | | --- | |  |  |  | | --- | | Beautiful CLI output (optional) | |

**📓 4. Launching Jupyter Notebook**

In your project folder:

|  |
| --- |
| jupyter notebook |

A browser will open. You can create ‘.ipynb’ files and run code cells interactively — great for learning and testing algorithms.

**You can create a ‘requirements.txt’to share your setup:**

|  |
| --- |
| pip freeze > requirements.txt |

**🛠️ 6. Optional Tools (Highly Recommended)**

|  |  |  |
| --- | --- | --- |
| **Tool** | **Use Case** | **Install Command** |
|  | Auto-code formatter | pip install black |
|  | Testing algorithms | pip install pytest |
|  | Enhanced interactive shell | pip install ipython |
|  | Advanced graph visualizations | see note below |

⚠️ pygraphviz may require additional system libraries:  
On Ubuntu:

|  |
| --- |
| sudo apt install graphviz libgraphviz-dev |

# Appendix – 05: Step-by-Step Guide of Various Algorithm with Python Code

## 01 – Implementation of Dijkstra’s Algorithm

## 02 - Implementation of Bellman-Ford Algorithm

## 03 - Implementation of Kahn’s Algorithm

## 04 - Implementation of Dinic’s Algorithm

## 05 - Implementation of Ford-Fulkerson Algorithm

## 06 - Implementation of Prim’s Algorithm

## 07 - Implementation of Kruskal’s Algorithm

## 08 - Implementation of Basic Operation Associated with B+ Tree

## 09 - Implementation of K – Dimensional Tree

## 10 - Implementation of Rabin-Krap Algorithm

🔍 **Rabin-Karp**[[8]](#footnote-8) **Algorithm: Overview**

The **Rabin-Karp algorithm**[[9]](#footnote-9)is a **string-searching algorithm** used to find a **pattern** in a **text** efficiently. It uses **hashing** to speed up the process of checking whether a substring of the text matches the pattern.

Instead of comparing each substring character-by-character, it compares **hash values** first. If the hash values match, then it does a direct comparison to confirm the match.

**🧠 Idea Behind Rabin-Karp**

1. Compute the **hash** of the pattern.
2. Compute the **hash** of all substrings of the text with the same length as the pattern.
3. Compare the hash of the pattern with each substring’s hash.
4. If hashes match, compare the actual substring and pattern to confirm a match (to avoid false positives due to hash collisions).

**🪜 Steps to Implement (Plain English)**

**Inputs:**

* *text* — the main string (e.g., a document or sentence)
* *pattern* — the string to search for (e.g., a word)

**Step-by-Step:**

1. Let *m* be the length of the pattern and n be the length of the text.
2. Choose a **base (d)** and a **prime number (q)** for hash calculations.
3. Calculate the **hash of the pattern**.
4. Calculate the **initial hash** of the first window (substring) of the text.
5. Slide the window over the text:
   * For each window, compare its hash with the pattern hash.
   * If they match, do a direct comparison of strings.
   * If not, move the window one character ahead and **update the hash** efficiently (rolling hash).
6. Repeat until you reach the end of the text.

🔢 **Pseudo Code**

|  |
| --- |
| function RabinKarp(text, pattern, d, q):  n ← length of text  m ← length of pattern  h ← (d^(m-1)) mod q // used to remove leading digit  p\_hash ← 0 // hash value for pattern  t\_hash ← 0 // hash value for current text window  // Precompute hash of pattern and first window  for i from 0 to m-1:  p\_hash ← (d \* p\_hash + ASCII(pattern[i])) mod q  t\_hash ← (d \* t\_hash + ASCII(text[i])) mod q  for i from 0 to n - m:  if p\_hash == t\_hash:  if text[i : i+m] == pattern:  print("Pattern found at index", i)  // Update hash for next window  if i < n - m:  t\_hash ← (d \* (t\_hash - ASCII(text[i]) \* h) + ASCII(text[i + m])) mod q  if t\_hash < 0:  t\_hash ← t\_hash + q |

**📝 Explanation of Important Parts**

* **Base (d)**: Usually 256 (number of possible characters like in ASCII).
* **Prime (q)**: A large prime number to reduce hash collisions.
* **Rolling hash**: Instead of recalculating the entire hash for each window, it uses:

|  |
| --- |
| new\_hash = d\*(old\_hash - leading\_char\*high\_pow) + new\_char |

* **Modulo operation** (mod q) ensures the hash value stays within a range and avoids overflow.
* **Hash collision**: Sometimes, two different strings can have the same hash. That’s why we double-check using actual string comparison when hashes match.

**✅ Example (In Plain Words)**

Let’s say:

* text = "ababcab"
* pattern = "abc"

**Steps:**

1. Compute hash of "abc" = p\_hash
2. Compute hash of first window "aba" in text
3. Slide window: "aba", "bab", "abc"... and so on
4. When hash matches:
   * Check if substring matches pattern
   * If yes, report the index

SOME MORE LEFT FROM “A step-by-step example with real values”

## 11 - Implementation of KMP Algorithm

## 12 - Implementation of Union by Rank Algorithm

## 13 - Implementation of Various Sorting Algorithm

## 14- Implementation of Quick Sort Algorithm

Quick Sort[[10]](#footnote-10) is a **Divide and Conquer** sorting algorithm. It works by picking a **pivot element**, and **partitioning** the array such that:

* All elements **less than the pivot** go to the **left** of it.
* All elements **greater than the pivot** go to the **right** of it.

This is done **recursively** on the left and right parts, until the entire array is sorted.

**🔍 Steps to Implement Quick Sort (Plain English)**

1. **Choose a Pivot Element**
   * Usually the **last** element, but it could also be the first or a random one.
2. **Partition the Array**
   * Arrange all elements so that:
     + Elements **less than the pivot** are on its **left**.
     + Elements **greater than the pivot** are on its **right**.
   * The pivot is now in its **correct sorted position**.
3. **Recursively Apply Quick Sort**
   * On the **left subarray** (before pivot)
   * On the **right subarray** (after pivot)
4. **Base Case:**
   * When subarray has **0 or 1 element**, it's already sorted.

**🧠 Key Idea: Partitioning**

* Partitioning is the core step. We move elements around so that after this step, the pivot is in its final place, and the rest are partially sorted.

📜 Pseudo Code for Quick Sort

|  |
| --- |
| function quickSort(arr, low, high)  if low < high  pivotIndex = partition(arr, low, high)  quickSort(arr, low, pivotIndex - 1)  quickSort(arr, pivotIndex + 1, high)  function partition(arr, low, high)  pivot = arr[high] // Choose pivot  i = low - 1 // Index of smaller element  for j = low to high - 1  if arr[j] <= pivot  i = i + 1  swap arr[i] with arr[j]  swap arr[i + 1] with arr[high]  return i + 1 // Return pivot index |

**🔎 Explanation of Pseudo Code**

**quickSort(arr, low, high)**

* This is the main recursive function.
* It keeps dividing the array into smaller parts until it's sorted.
* partition() finds the correct index of the pivot and rearranges the array accordingly.

**partition(arr, low, high)**

* Chooses a pivot (here, the last element).
* i tracks the last position where a smaller-than-pivot element was placed.
* The loop checks each element:
  + If it’s smaller than or equal to the pivot, we increase i and swap the element with arr[i].
* Finally, put the pivot in its correct place (arr[i+1]).

⚖️ Time and Space Complexity

|  |  |  |
| --- | --- | --- |
| **Case** | **Time Complexity** | **When** |
| Best |  |  |
| Average |  |  |
| Worst |  |  |

* **Space Complexity:** O(log n) for recursion stack (in-place sort)

🔢 Unsorted Array

|  |
| --- |
| [25, 17, 31, 13, 2, 40, 55, 29, 10, 22, 18] |

We'll use the **last element as pivot** in each partition step.

**🧩 Step-by-Step Quick Sort**

**🎯 Step 1: quickSort(arr, 0, 10)**

* **Array:** [25, 17, 31, 13, 2, 40, 55, 29, 10, 22, 18], **Pivot:** 18

**🔄 Partition Step:**

1. Start i = -1
2. Traverse j = 0 to 9:
   * If arr[j] <= 18, increment i, swap arr[i] and arr[j]
3. After partitioning:
   * Swaps result in → [17, 13, 2, 10, 18, 40, 55, 29, 31, 25, 22]
   * Pivot 18 is placed at index 4.

✅ Left part (less than 18): [17, 13, 2, 10]  
✅ Right part (greater than 18): [40, 55, 29, 31, 25, 22]

**🔁 Step 2: quickSort(arr, 0, 3)**

* Subarray: [17, 13, 2, 10], **Pivot:** 10

**🔄 Partition Step:**

1. i = -1
2. Traverse j = 0 to 2
   * Compare each element with 10
3. Result after partition: [2, 10, 13, 17]
   * Pivot 10 is at index 1

Now:

* Left: [2] (base case, no further sort needed)
* Right: [13, 17]

**🔁 Step 3: quickSort(arr, 2, 3)**

* Subarray: [13, 17], **Pivot:** 17

**🔄 Partition Step:**

1. i = 1
2. j = 2 → 13 < 17 → swap (no real change)
3. Result: [2, 10, 13, 17]
   * Pivot 17 is at index 3 (already sorted)

✅ Now left side of main array is sorted: [2, 10, 13, 17]

**🔁 Step 4: quickSort(arr, 5, 10)**

* Subarray: [40, 55, 29, 31, 25, 22], **Pivot:** 22

**🔄 Partition Step:**

1. i = 4
2. Traverse j = 5 to 9
   * Compare each element with 22
3. After partition:
   * Swap 22 into correct place → [2, 10, 13, 17, 18, 22, 55, 29, 31, 25, 40]
   * Pivot 22 is at index 5

✅ Left of pivot: none  
✅ Right: [55, 29, 31, 25, 40]

**🔁 Step 5: quickSort(arr, 6, 10)**

* Subarray: [55, 29, 31, 25, 40] **Pivot:** 40

**🔄 Partition Step:**

1. i = 5
2. Traverse j = 6 to 9
   * Compare with 40
3. After partition:
   * [2, 10, 13, 17, 18, 22, 29, 31, 25, 40, 55]
   * Pivot 40 is placed at index 9

✅ Left: [29, 31, 25]  
✅ Right: [55] (base case)

**🔁 Step 6: quickSort(arr, 6, 8)**

* Subarray: [29, 31, 25]
* **Pivot:** 25

**🔄 Partition Step:**

1. i = 5
2. Compare elements with 25
3. Swap 25 into place → [2, 10, 13, 17, 18, 22, 25, 31, 29, 40, 55]

✅ Pivot at index 6  
✅ Left: none  
✅ Right: [31, 29]

**🔁 Step 7: quickSort(arr, 7, 8)**

* Subarray: [31, 29]
* **Pivot:** 29

**🔄 Partition Step:**

1. Swap 29 into place → [2, 10, 13, 17, 18, 22, 25, 29, 31, 40, 55]

✅ Sorted!

✅ Final Sorted Array:

|  |
| --- |
| [2, 10, 13, 17, 18, 22, 25, 29, 31, 40, 55] |

📌 Summary

| **Step** | **Subarray** | **Pivot** | **Result After Partition** |
| --- | --- | --- | --- |
| 1 | Entire Array | 18 | Pivot at index 4 |
| 2 | [17, 13, 2, 10] | 10 | Pivot at index 1 |
| 3 | [13, 17] | 17 | Sorted |
| 4 | [40, 55, 29, 31, 25, 22] | 22 | Pivot at index 5 |
| 5 | [55, 29, 31, 25, 40] | 40 | Pivot at index 9 |
| 6 | [29, 31, 25] | 25 | Pivot at index 6 |
| 7 | [31, 29] | 29 | Pivot at index 7 |

🐍 Python Code

|  |
| --- |
| def quick\_sort(arr, low, high):  if low < high:  # Partition the array and get the pivot index  pivot\_index = partition(arr, low, high)    # Recursively apply quick sort to the left and right subarrays  quick\_sort(arr, low, pivot\_index - 1)  quick\_sort(arr, pivot\_index + 1, high)  def partition(arr, low, high):  pivot = arr[high] # Pivot element  i = low - 1 # Index of smaller element  for j in range(low, high):  if arr[j] <= pivot:  i += 1  arr[i], arr[j] = arr[j], arr[i] # Swap  # Place pivot in correct position  arr[i + 1], arr[high] = arr[high], arr[i + 1]  return i + 1  # Example usage  arr = [25, 17, 31, 13, 2, 40, 55, 29, 10, 22, 18]  quick\_sort(arr, 0, len(arr) - 1)  print("Sorted array:", arr) |

💻 C Code

|  |
| --- |
| #include <stdio.h>  void swap(int\* a, int\* b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  int partition(int arr[], int low, int high) {  int pivot = arr[high]; // Pivot  int i = low - 1; // Index of smaller element  for (int j = low; j <= high - 1; j++) {  if (arr[j] <= pivot) {  i++;  swap(&arr[i], &arr[j]); // Swap  }  }  swap(&arr[i + 1], &arr[high]); // Place pivot correctly  return (i + 1);  }  void quick\_sort(int arr[], int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);  // Recursively sort subarrays  quick\_sort(arr, low, pi - 1);  quick\_sort(arr, pi + 1, high);  }  }  int main() {  int arr[] = {25, 17, 31, 13, 2, 40, 55, 29, 10, 22, 18};  int n = sizeof(arr) / sizeof(arr[0]);  quick\_sort(arr, 0, n - 1);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  return 0;  } |

**Q**. Implement a new Version of Quick sort using 2 pointers from either end to an array, ‘i < pivot’ insert using begin, likewise ‘I > pivot’ insert using last. Pivot can be inserted at the last using the condition ‘begin pointer + 1 = last pointer’.

* 1. Is this version Stable
  2. Is this version In-place
  3. Time and Space complexity of Such

## 15- Implementation of Merge Sort Algorithm

**Merge Sort**[[11]](#footnote-11) is a **divide-and-conquer** algorithm. The main idea is to **divide the array into halves**, **sort** each half, and then **merge** them back together in sorted order.

**🔧 Steps to Implement Merge Sort (in Plain English):**

1. **If the array has 0 or 1 elements**, it is already sorted. Just return it.
2. **Divide** the array into two roughly equal halves.
3. **Recursively apply** merge sort to both halves.
4. **Merge** the two sorted halves into one sorted array:
   * Compare the first elements of both halves.
   * Pick the smaller one and add it to a new array.
   * Repeat this process until one half is empty.
   * Add all remaining elements from the non-empty half.

🧠 Pseudocode for Merge Sort:

|  |
| --- |
| MERGE\_SORT(arr):  if length of arr <= 1:  return arr  mid = length of arr // 2  left\_half = MERGE\_SORT(arr[0:mid])  right\_half = MERGE\_SORT(arr[mid:end])  return MERGE(left\_half, right\_half)  MERGE(left, right):  result = empty array  while left and right are not empty:  if left[0] <= right[0]:  append left[0] to result  remove left[0] from left  else:  append right[0] to result  remove right[0] from right  append any remaining elements in left to result  append any remaining elements in right to result  return result |

Let’s take an unsorted array of 11 numbers:

|  |
| --- |
| [38, 12, 45, 22, 9, 18, 7, 30, 50, 5, 27] |

We will apply **Merge Sort** step-by-step, showing all **splits**, **recursions**, and **merges**.

🔁 Step 1: Divide the array into halves recursively

Split it:

|  |
| --- |
| Left: [38, 12, 45, 22, 9]  Right: [18, 7, 30, 50, 5, 27] |

**🔁 Step 2: Keep splitting the left part**

**Left: [38, 12, 45, 22, 9]**

Split → [38, 12] and [45, 22, 9]

**[38, 12] → Split further → [38] and [12] → Merge → [12, 38]**

**[45, 22, 9] → Split → [45] and [22, 9]**

* [22, 9] → Split → [22] and [9] → Merge → [9, 22]
* Now merge [45] and [9, 22] → [9, 22, 45]

Now merge [12, 38] and [9, 22, 45]:

|  |
| --- |
| Compare and Merge: [12, 38] and [9, 22, 45]  - 9 < 12 → [9]  - 12 < 22 → [9, 12]  - 22 < 38 → [9, 12, 22]  - 38 < 45 → [9, 12, 22, 38]  - Add remaining 45 → [9, 12, 22, 38, 45] |

Left part is now sorted: [9, 12, 22, 38, 45]

**🔁 Step 3: Split and sort the right part**

**Right: [18, 7, 30, 50, 5, 27]**

Split → [18, 7, 30] and [50, 5, 27]

**[18, 7, 30] → Split → [18] and [7, 30]**

* [7, 30] → Split → [7] and [30] → Merge → [7, 30]
* Now merge [18] and [7, 30] → [7, 18, 30]

**[50, 5, 27] → Split → [50] and [5, 27]**

* [5, 27] → Split → [5] and [27] → Merge → [5, 27]
* Now merge [50] and [5, 27] → [5, 27, 50]

Now merge [7, 18, 30] and [5, 27, 50]:

|  |
| --- |
| Compare and Merge: [7, 18, 30] and [5, 27, 50]  - 5 < 7 → [5]  - 7 < 27 → [5, 7]  - 18 < 27 → [5, 7, 18]  - 27 < 30 → [5, 7, 18, 27]  - 30 < 50 → [5, 7, 18, 27, 30]  - Add 50 → [5, 7, 18, 27, 30, 50] |

**Right part is now sorted: [5, 7, 18, 27, 30, 50]**

**🔁 Step 4: Final Merge**

Merge Left [9, 12, 22, 38, 45] and Right [5, 7, 18, 27, 30, 50]:

|  |
| --- |
| Compare and Merge: [9, 12, 22, 38, 45] and [5, 7, 18, 27, 30, 50]  - 5 < 9 → [5]  - 7 < 9 → [5, 7]  - 9 < 18 → [5, 7, 9]  - 12 < 18 → [5, 7, 9, 12]  - 18 < 22 → [5, 7, 9, 12, 18]  - 22 < 27 → [5, 7, 9, 12, 18, 22]  - 27 < 38 → [5, 7, 9, 12, 18, 22, 27]  - 30 < 38 → [5, 7, 9, 12, 18, 22, 27, 30]  - 38 < 50 → [5, 7, 9, 12, 18, 22, 27, 30, 38]  - 45 < 50 → [5, 7, 9, 12, 18, 22, 27, 30, 38, 45]  - Add remaining 50 → [5, 7, 9, 12, 18, 22, 27, 30, 38, 45, 50] |

✅ Final Sorted Array:

|  |
| --- |
| [5, 7, 9, 12, 18, 22, 27, 30, 38, 45, 50] |

🐍 Python Code

|  |
| --- |
| def merge\_sort(arr):  if len(arr) <= 1:  return arr  mid = len(arr) // 2  left = merge\_sort(arr[:mid])  right = merge\_sort(arr[mid:])  return merge(left, right)  def merge(left, right):  result = []  i = j = 0  # Compare and merge  while i < len(left) and j < len(right):  if left[i] <= right[j]:  result.append(left[i])  i += 1  else:  result.append(right[j])  j += 1  # Add remaining elements  result.extend(left[i:])  result.extend(right[j:])  return result  # Example array  arr = [38, 12, 45, 22, 9, 18, 7, 30, 50, 5, 27]  sorted\_arr = merge\_sort(arr)  print("Sorted Array:", sorted\_arr) |

💻 C Code

|  |
| --- |
| #include <stdio.h>  void merge(int arr[], int left, int mid, int right) {  int i = left, j = mid + 1, k = 0;  int temp[right - left + 1];  // Merge the two halves  while (i <= mid && j <= right) {  if (arr[i] <= arr[j])  temp[k++] = arr[i++];  else  temp[k++] = arr[j++];  }  // Copy remaining elements  while (i <= mid)  temp[k++] = arr[i++];  while (j <= right)  temp[k++] = arr[j++];  // Copy back to original array  for (i = left, k = 0; i <= right; i++, k++)  arr[i] = temp[k];  }  void mergeSort(int arr[], int left, int right) {  if (left >= right)  return;  int mid = (left + right) / 2;  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  merge(arr, left, mid, right);  }  int main() {  int arr[] = {38, 12, 45, 22, 9, 18, 7, 30, 50, 5, 27};  int n = sizeof(arr) / sizeof(arr[0]);  mergeSort(arr, 0, n - 1);  printf("Sorted Array: ");  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  return 0;  } |

## 16 - Implementation of Heap Sort Algorithm

**Heap Sort**[[12]](#footnote-12) is a **comparison-based sorting algorithm** that uses a special binary tree structure called a **heap**. It works in two main phases:

**🔧 Steps to Implement Heap Sort (in Plain English)**

1. **Build a Max Heap** from the input data.
   * A **Max Heap** is a binary tree where the parent is **always greater than** its children.
   * The largest element will be at the root (index 0 in array).
2. **Extract the maximum element** (which is at the root).
   * Swap it with the last item in the heap.
   * Shrink the heap by 1 (exclude the last item).
   * Heapify the root to maintain the max-heap property.
3. **Repeat step 2** until the heap is empty.

📋 **Pseudo Code of Heap Sort**

|  |
| --- |
| HEAPSORT(arr):  n = length of arr  // Step 1: Build Max Heap  for i = n/2 - 1 down to 0:  HEAPIFY(arr, n, i)  // Step 2: Extract elements one by one  for i = n-1 down to 1:  swap arr[0] and arr[i] // move current root to end  HEAPIFY(arr, i, 0) // heapify reduced heap  HEAPIFY(arr, n, i):  largest = i // initialize largest as root  left = 2\*i + 1 // left child  right = 2\*i + 2 // right child  if left < n and arr[left] > arr[largest]:  largest = left  if right < n and arr[right] > arr[largest]:  largest = right  if largest != i:  swap arr[i] and arr[largest]  HEAPIFY(arr, n, largest) |

**🔄 Explanation of Pseudo Code**

* The first loop (in HEAPSORT) builds the **max-heap**. We start from the last non-leaf node (n/2 - 1) and move upward.
* In the second loop, we repeatedly:
  + Move the root (max element) to the end of the array,
  + Reduce the size of the heap, and
  + Re-heapify the remaining heap using HEAPIFY.

The HEAPIFY function ensures the heap property is maintained:

* Compares a node with its children.
* If a child is larger, it swaps and recursively heapifies the affected subtree.

**🧠 Memory and Time Complexity**

* **Time Complexity:**
  + Building Heap: O(n)
  + Heapify for n elements: O(n log n)
  + **Total: O(n log n)**
* **Space Complexity: O(1)** (in-place sorting)

Let’s take an **unsorted array** of 11 elements:

|  |
| --- |
| Initial array: [9, 4, 7, 1, -2, 6, 5, 0, -1, 3, 2] |

We’ll now go step-by-step through **Heap Sort**, including:

1. **Building a Max Heap**
2. **Sorting the array by repeatedly extracting the max**

**📘 Step 1: Build a Max Heap**

We use HEAPIFY from the last non-leaf node up to the root.

Total elements = 11  
Last non-leaf index = ⌊11/2⌋ - 1 = 4

**🔄 Start heapifying from index 4 to 0**

**i = 4 → element = -2**

Children: left = 9 (value = 3), right = 10 (value = 2)  
Max = 3 → swap -2 and 3

|  |
| --- |
| After heapify(4): [9, 4, 7, 1, 3, 6, 5, 0, -1, -2, 2] |

**i = 3 → element = 1**

Children: 7 (value = 0), 8 (value = -1)  
1 > both → no change

|  |
| --- |
| After heapify(3): [9, 4, 7, 1, 3, 6, 5, 0, -1, -2, 2] |

**i = 2 → element = 7**

Children: 5 (6), 6 (5)  
7 > both → no change

|  |
| --- |
| After heapify(2): [9, 4, 7, 1, 3, 6, 5, 0, -1, -2, 2] |

**i = 1 → element = 4**

Children: 3 (1), 4 (3)  
4 > both → no change

|  |
| --- |
| After heapify(1): [9, 4, 7, 1, 3, 6, 5, 0, -1, -2, 2] |

**i = 0 → element = 9**

Children: 1 (4), 2 (7)  
9 > both → no change

|  |
| --- |
| After heapify(0): [9, 4, 7, 1, 3, 6, 5, 0, -1, -2, 2] |

✅ Max Heap built:

|  |
| --- |
| [9, 4, 7, 1, 3, 6, 5, 0, -1, -2, 2] |

**📘 Step 2: Sort the Array**

Repeatedly extract the root (maximum), place it at the end, and **heapify** the reduced heap.

We will show all steps:

**Pass 1:**

* Swap root with last → swap 9 and 2
* Array: [2, 4, 7, 1, 3, 6, 5, 0, -1, -2, 9]
* Heapify root (0) in size 10

|  |
| --- |
| * 2 → children 4 and 7 → max = 7 at index 2 → swap * 7 → children 6 and 5 → max = 6 → no change needed |

Result:

|  |
| --- |
| [7, 4, 2, 1, 3, 6, 5, 0, -1, -2, 9] |

**Pass 2:**

* Swap root (7) with 10th (−2)
* [−2, 4, 2, 1, 3, 6, 5, 0, -1, 7, 9]
* Heapify 0 in size 9

|  |
| --- |
| * −2 → children 4, 2 → max = 4 → swap * 4 → children 1, 3 → max = 3 → swap * 3 → no children → done |

Result:

|  |
| --- |
| [4, 3, 2, 1, -2, 6, 5, 0, -1, 7, 9] |

**Pass 3:**

* Swap 4 and −1
* [−1, 3, 2, 1, -2, 6, 5, 0, 4, 7, 9]
* Heapify size 8

|  |
| --- |
| * −1 → children 3, 2 → max = 3 → swap * 3 → children 1, -2 → max = 1 → swap * 1 → no children → done |

Result:

|  |
| --- |
| [3, 1, 2, -1, -2, 6, 5, 0, 4, 7, 9] |

**Pass 4:**

* Swap 3 and 0
* [0, 1, 2, -1, -2, 6, 5, 3, 4, 7, 9]
* Heapify size 7

|  |
| --- |
| * 0 → children 1 and 2 → max = 2 → swap * 2 → no bigger children → done |

Result:

|  |
| --- |
| [2, 1, 0, -1, -2, 6, 5, 3, 4, 7, 9] |

**Pass 5:**

* Swap 2 and 5
* [5, 1, 0, -1, -2, 6, 2, 3, 4, 7, 9]
* Heapify size 6

|  |
| --- |
| 5 → children 1, 0 → no change → max already at root |

Result:

|  |
| --- |
| [5, 1, 0, -1, -2, 6, 2, 3, 4, 7, 9] |

**Pass 6:**

* Swap 5 and 6
* [6, 1, 0, -1, -2, 5, 2, 3, 4, 7, 9]
* Heapify size 5

|  |
| --- |
| 6 → larger than children → done |

**Pass 7:**

* Swap 6 and −2
* [−2, 1, 0, -1, 6, 5, 2, 3, 4, 7, 9]
* Heapify size 4

|  |
| --- |
| * −2 → children 1, 0 → max = 1 → swap * 1 → no bigger children → done |

Result:

|  |
| --- |
| [1, -2, 0, -1, 6, 5, 2, 3, 4, 7, 9] |

**Pass 8:**

* Swap 1 and -1
* [-1, -2, 0, 1, 6, 5, 2, 3, 4, 7, 9]
* Heapify size 3

|  |
| --- |
| -1 → children -2, 0 → max = 0 → swap |

Result:

|  |
| --- |
| [0, -2, -1, 1, 6, 5, 2, 3, 4, 7, 9] |

**Pass 9:**

* Swap 0 and -2
* [-2, 0, -1, 1, 6, 5, 2, 3, 4, 7, 9]
* Heapify size 2

|  |
| --- |
| -2 → child 0 → swap |

**Pass 10:**

* Swap 0 and -2 → done

✅ Final Sorted Array:

|  |
| --- |
| [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 9] |

🐍 Python Code for Heap Sort

|  |
| --- |
| def heapify(arr, n, i):  largest = i # Initialize largest as root  left = 2 \* i + 1 # left = 2\*i + 1  right = 2 \* i + 2 # right = 2\*i + 2  # If left child exists and is greater than root  if left < n and arr[left] > arr[largest]:  largest = left  # If right child exists and is greater than largest so far  if right < n and arr[right] > arr[largest]:  largest = right  # If largest is not root  if largest != i:  arr[i], arr[largest] = arr[largest], arr[i] # swap  heapify(arr, n, largest) # Recursively heapify the affected sub-tree  def heap\_sort(arr):  n = len(arr)  # Build a maxheap (rearrange array)  for i in range(n // 2 - 1, -1, -1):  heapify(arr, n, i)  # Extract elements one by one  for i in range(n - 1, 0, -1):  arr[0], arr[i] = arr[i], arr[0] # swap  heapify(arr, i, 0)  # Test  arr = [9, 4, 7, 1, -2, 6, 5, 0, -1, 3, 2]  heap\_sort(arr)  print("Sorted array is:", arr) |

🔧 C Code for Heap Sort

|  |
| --- |
| #include <stdio.h>  // To heapify a subtree rooted with node i which is an index in arr[]  void heapify(int arr[], int n, int i) {  int largest = i; // Initialize largest as root  int left = 2 \* i + 1; // left = 2\*i + 1  int right = 2 \* i + 2; // right = 2\*i + 2  // If left child exists and is greater than root  if (left < n && arr[left] > arr[largest])  largest = left;  // If right child exists and is greater than largest so far  if (right < n && arr[right] > arr[largest])  largest = right;  // If largest is not root  if (largest != i) {  // Swap  int temp = arr[i];  arr[i] = arr[largest];  arr[largest] = temp;  // Recursively heapify the affected sub-tree  heapify(arr, n, largest);  }  }  // Main function to perform heap sort  void heapSort(int arr[], int n) {  // Build max heap (rearrange array)  for (int i = n / 2 - 1; i >= 0; i--)  heapify(arr, n, i);  // Extract elements from heap one by one  for (int i = n - 1; i > 0; i--) {  // Move current root to end  int temp = arr[0];  arr[0] = arr[i];  arr[i] = temp;  // call max heapify on the reduced heap  heapify(arr, i, 0);  }  }  // Utility function to print array  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  }  // Driver code  int main() {  int arr[] = {9, 4, 7, 1, -2, 6, 5, 0, -1, 3, 2};  int n = sizeof(arr) / sizeof(arr[0]);  heapSort(arr, n);  printf("Sorted array is:\n");  printArray(arr, n);  return 0;  } |

# Appendix – 06: Working with Graph using NetworkX

(https://chatgpt.com/c/68490260-0394-800c-a581-9d6389235c43)

**🧾 Introduction**

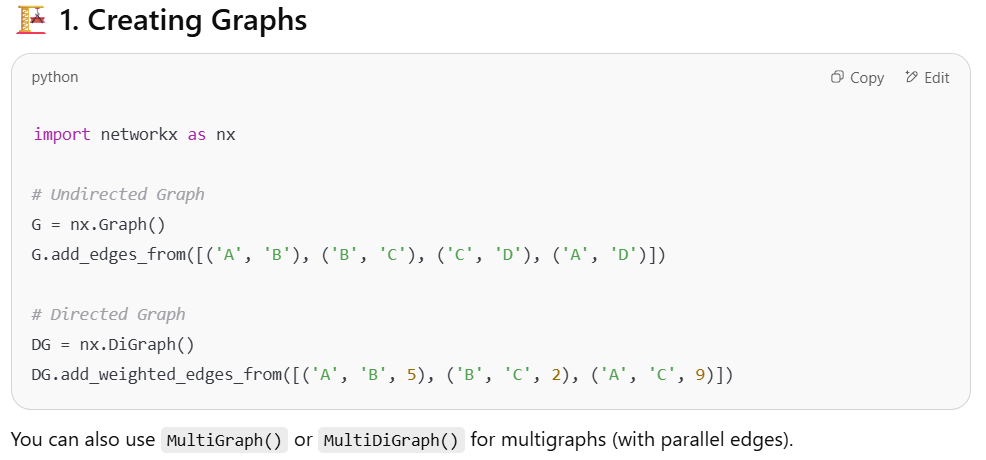
Graphs are fundamental in computer science and algorithm design. They model relationships between entities—like cities on a map, web pages, social networks, and more.

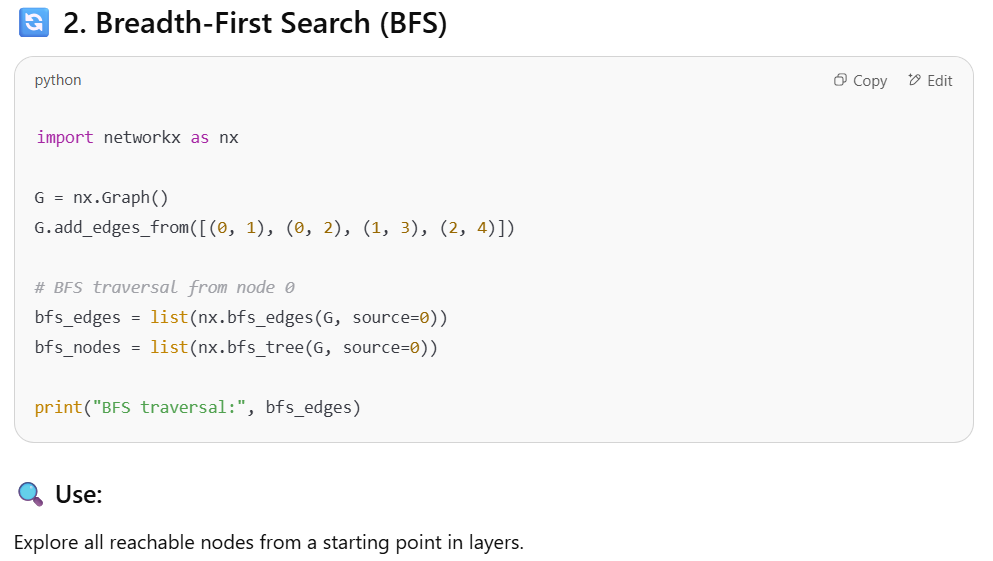
This appendix introduces [**NetworkX**](https://networkx.org/)(https://networkx.org/)—a Python library for creating, manipulating, and visualizing complex networks—to help you **experiment interactively** with graph algorithms while learning them.

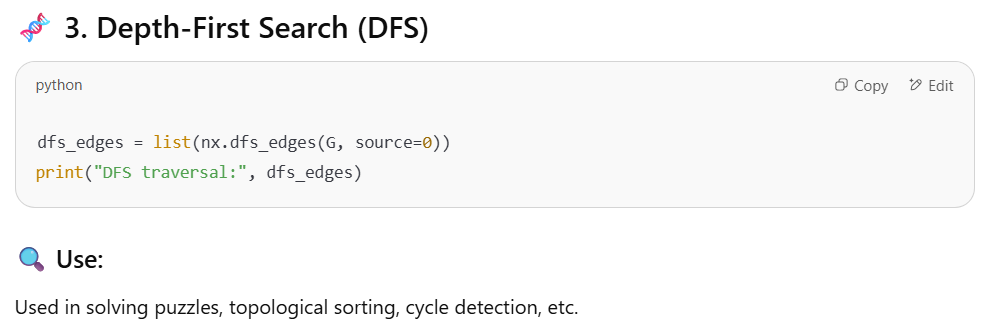
**✅ Why** [**NetworkX**](https://networkx.org/) **is Useful**

* Easy to create and visualize graphs
* Supports directed, undirected, weighted, and multigraphs
* Built-in implementations of many classic graph algorithms
* Useful for both **learning concepts** and **experimenting interactively**

**Here are some examples, please develop some more by yourself.**

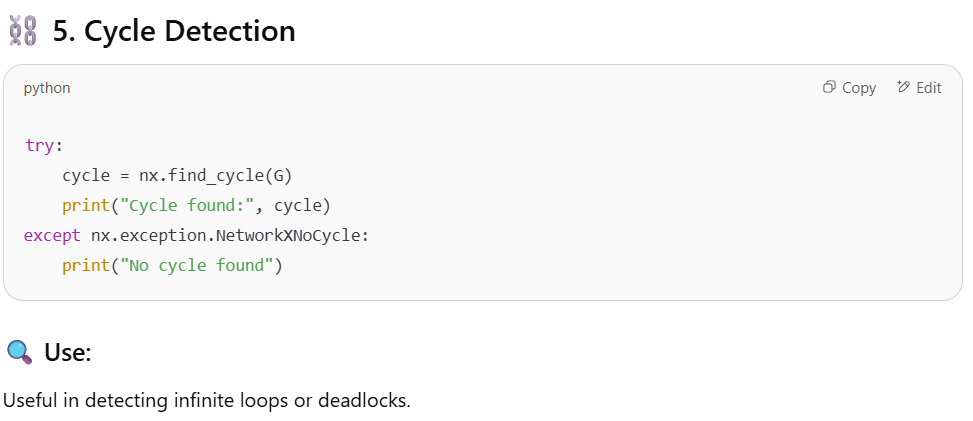


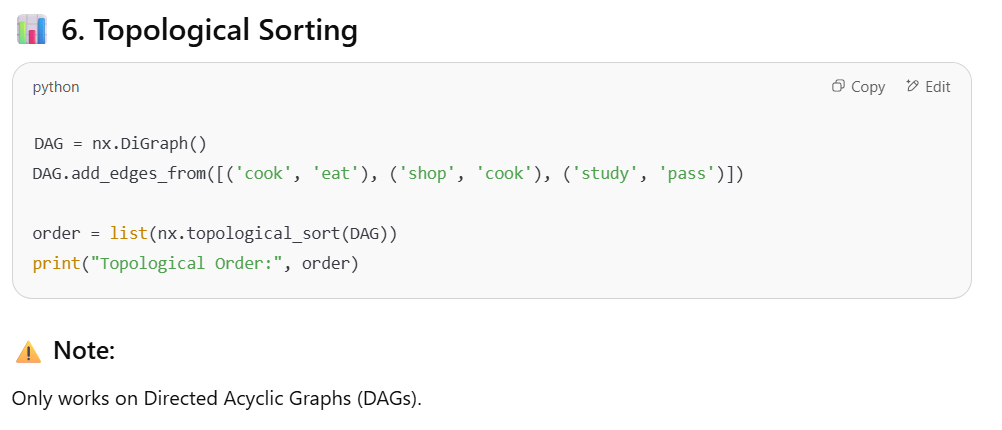


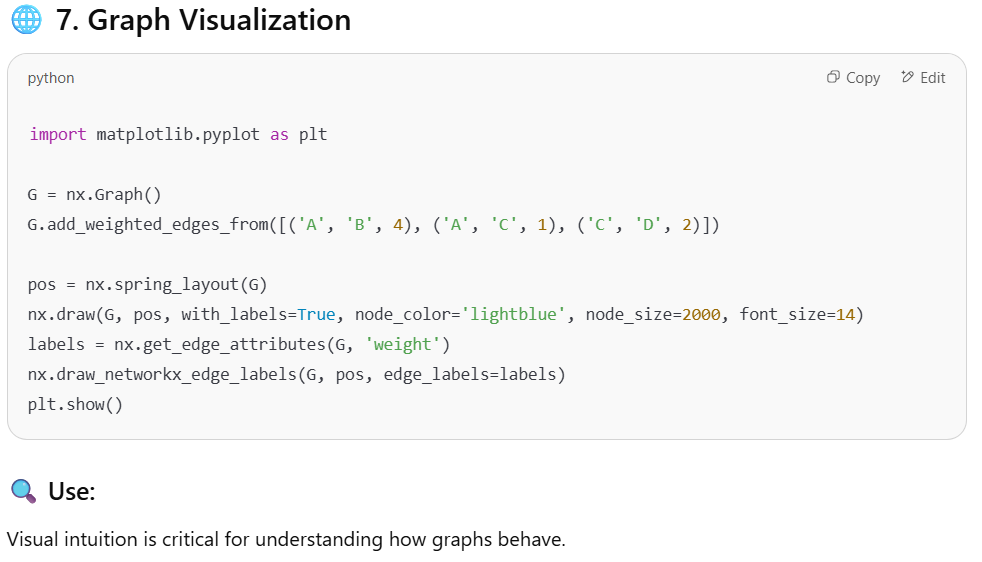






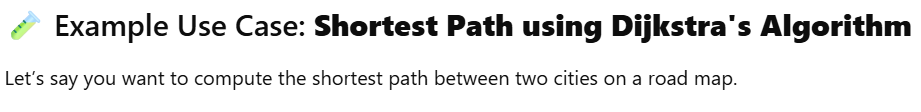












|  |
| --- |
| import networkx as nx  import matplotlib.pyplot as plt  # Step 1: Create a directed weighted graph  G = nx.DiGraph()  # Step 2: Add edges (node1, node2, weight)  G.add\_weighted\_edges\_from([  ('A', 'B', 4),  ('A', 'C', 2),  ('B', 'C', 5),  ('B', 'D', 10),  ('C', 'D', 3)  ])  # Step 3: Compute shortest path from A to D  path = nx.dijkstra\_path(G, source='A', target='D')  length = nx.dijkstra\_path\_length(G, source='A', target='D')  print("Shortest path:", path)  print("Path length:", length)  # Step 4: Visualize the graph  pos = nx.spring\_layout(G)  nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=2000, font\_size=16)  edge\_labels = nx.get\_edge\_attributes(G, 'weight')  nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels)  plt.show() |

# Appendix – 07: Essential Problems from CLRS

**🧠 Essential Problems from CLRS (by Chapter)**

**Chapter 2: Getting Started**

* Insertion Sort (2.1)
* Merge Sort – including loop invariants (2.3)
* Binary Search (Exercise 2.3-5)
* Inversions in an array (Problem 2-4)

**Chapter 3: Growth of Functions**

* Asymptotic notation comparison problems
* Exercise 3.1-1 to 3.1-6 – proving O, Θ, and Ω relationships
* Exercise 3.2-3 – use of limits in asymptotic behaviour

**Chapter 4: Divide-and-Conquer**

* Maximum Subarray Problem (4.1)
* Recurrence Tree Method and Master Theorem (4.3)
* Strassen’s Matrix Multiplication (4.2)

**Chapter 6: Heapsort**

* Build-Max-Heap and Max-Heapify (6.3)
* Implement Priority Queue with Heap
* Median maintenance with two heaps (advanced)

**Chapter 7–8: Quicksort & Sorting Lower Bounds**

* Randomized Quicksort (7.3)
* Worst-case for Quicksort (Problem 7-1)
* Counting Sort and Radix Sort (8.2, 8.3)
* Lower bounds for comparison sorts (8.1)

**Chapter 9: Medians and Order Statistics**

* Randomized-Select algorithm (9.2)
* Deterministic Select – Median of Medians (9.3)

**Chapter 10–11: Elementary Data Structures & Hashing**

* Stack, Queue, Linked List operations (10.1–10.3)
* Hash Table with chaining and open addressing (11.2–11.4)
* Universal hashing (11.3)

**Chapter 12–13: Binary Search Trees & Red-Black Trees**

* In-order Traversal
* Search, Min, Max, Successor, Predecessor (12.2)
* Insert and Delete in BST
* Red-Black Tree Insertion & Deletion (13.3)

**Chapter 15: Dynamic Programming**

* Matrix Chain Multiplication (15.2)
* Longest Common Subsequence (15.4)
* Rod Cutting (15.1)
* Optimal BST (15.5, advanced)

**Chapter 16: Greedy Algorithms**

* Activity Selection Problem (16.1)
* Huffman Coding (16.3)
* Fractional Knapsack (Problem 16-1)

**Chapter 22–24: Graph Algorithms**

* BFS and DFS (22.2, 22.3)
* Topological Sort (22.4)
* Strongly Connected Components (22.5)
* Dijkstra’s Algorithm (24.3)
* Bellman-Ford Algorithm (24.1)
* Floyd-Warshall Algorithm (25.2)
* Minimum Spanning Trees: Prim’s and Kruskal’s (23.1, 23.2)

**Chapter 26–27: Max Flow**

* Ford-Fulkerson Algorithm (26.2)
* Bipartite Matching using flow (26.1, 26.3)
* Push-Relabel Algorithm (27.2)

1. “Introduction to Algorithm – CLRS, 4TH ed”, Page 102 [↑](#footnote-ref-1)
2. “Introduction to Algorithm – CLRS, 4TH ed”, Page 104 [↑](#footnote-ref-2)
3. “Introduction to Algorithm – CLRS, 4TH ed”, Page 105 [↑](#footnote-ref-3)
4. “Introduction to Algorithm – CLRS, 4TH ed”, Page 106 [↑](#footnote-ref-4)
5. “Introduction to Algorithm – CLRS, 4TH ed”, Page 76 [↑](#footnote-ref-5)
6. https://chatgpt.com/c/685143fb-437c-800c-952c-551b01ec216b [↑](#footnote-ref-6)
7. https://onlinecourses.nptel.ac.in/noc23\_cs88/unit?unit=17&assessment=154 [↑](#footnote-ref-7)
8. https://chatgpt.com/c/68513368-4bb0-800c-8173-e6a73909a047 [↑](#footnote-ref-8)
9. “Introduction to Algorithm – CLRS, 4TH ed”, Page 962 [↑](#footnote-ref-9)
10. https://chatgpt.com/c/6850e100-d448-800c-a1de-8eac43519715 [↑](#footnote-ref-10)
11. https://chatgpt.com/c/6850e962-4554-800c-9ec4-45559e5746a5 [↑](#footnote-ref-11)
12. https://chatgpt.com/c/6850ed1f-9e9c-800c-a6a1-3c77b1ccc736 [↑](#footnote-ref-12)