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

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

The Problem of Sorting

Input:- A sequence of $< a_1, a_2, ..., a_n >$ of numbers
Output:- A permutation of $< a_1', a_2', ..., a_n' >$ such that

$$a_1' \leq a_2' \leq .... \leq a_n$$

Example

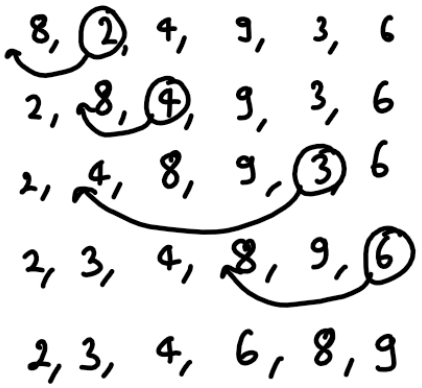
Input: 9 3 5 0 4 7
Output: 0 3 4 5 7 9

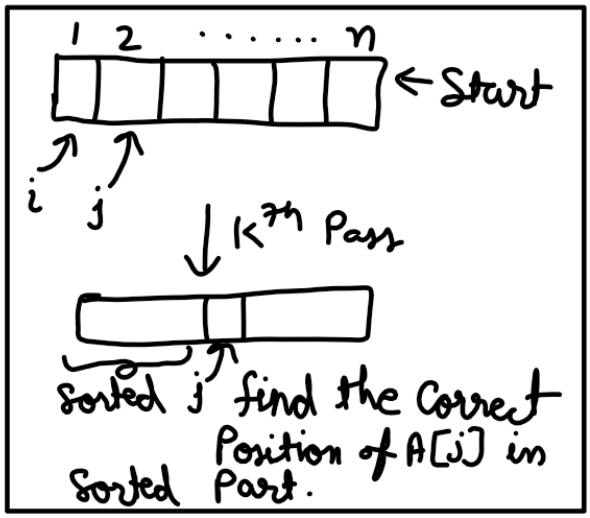
Pseudo Code:- Insertion Sort

```
Insertion Sort(A,n):
    for j ← 1 to n:
        do Key ← A[i]
           i ← j-1
           While i>0 & A[i] > key
               do A[i+1] ← A[i]
                  i ← i-1
           A[i+1] = key
```

Do a Dry Run of the code.

Example 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 (θ)

★ 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.

★ 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

Ignore machine-dependent constants.
Look at 'growth' of T(n) as $n \to \infty$

ASYMPTOTIC ANALYSIS

★ θ Notation

Maths:-

- $\boldsymbol{\theta(g(n))} = \{f(n) : \exists\ positive\ constant\ c_1, c_2$ and n_0 such that $0 \le \mathbf{c_1} \cdot g(n) \le f(n) \le \mathbf{c_2} \cdot g(n), \forall\ n \ge n_0\}$

Engineering:-

- Drop low order terms; ignore leading constants

Example:-

$3n - 90n + 5n - 1024 = \theta(n^3)$

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

★ O Notation

$O(g(n)) = \{f(n)$: ∃ positive constant c and n_0 such that $f(n) \le c * g(n)\ \forall\ n \ge n_0\}$

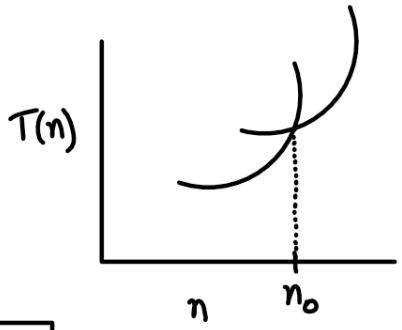
★ Ω Notation

$\Omega(g(n)) = \{f(n)$: ∃ positive constant c and n_0 such that $f(n) \ge c * g(n)\ \forall\ n \ge n_0\}$

★ **Asymptotic Notation**

When n gets large enough a $\theta(n^2)$ algorithm always beats a $O(n^3)$ 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**

- **Worst Case:** Input Inversely sorted.

$$T(n) = \sum_{j=2}^{n} \theta(j) = \theta(n^2) \text{ [Arithmetic Series]}$$

- **Average Case:** All permutation equally likely.

$$T(n) = \sum_{j=2}^{n} \theta\left(\frac{j}{2}\right) = \theta(n^2)$$

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

Lecture 04: Recurrence for Merge Sort

Topics to be Covered: -

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

Merge Sort

MERGE_SORT A[1 .... n]

To sort n numbers

1. If n = 1, done
2. Recursively Sort A[1 .... n/2] and A[⌈n/2⌉+1 .... n]
3. Merge the two sorted lists

Key Sub-Routine : MERGE

↳ Time = Θ(n) for n input

★ Analysis of Merge Sort

MERGE_SORT $A[1 \ldots n]$

$T(n)$	To sort n numbers
$\Theta(1)$	1. If $n = 1$, done
$\Theta(n/2)$	2. Recursively Sort $A[1 \ldots n/2]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$
$\Theta(n)$	3. Merge the two sorted lists

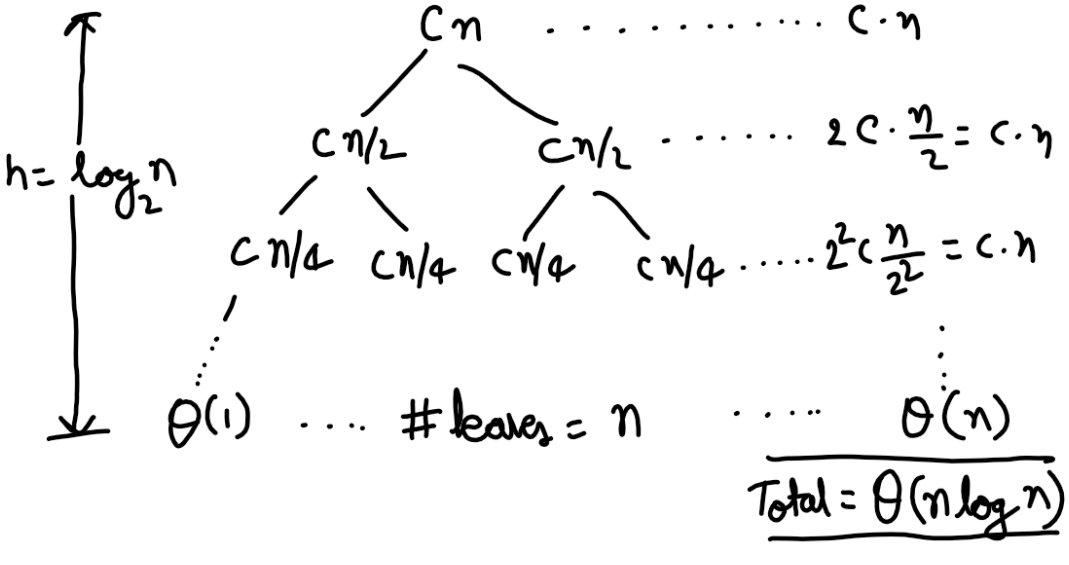
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$$

★ Recurrence for Merge Sort

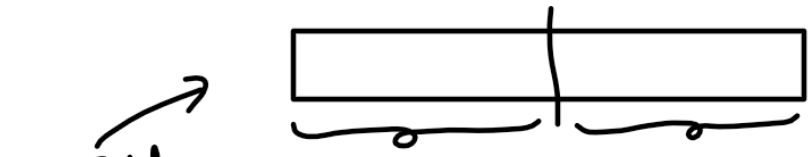
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

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

★ Recursion Tree



★ **Best case of Merge Sort**



Not Inplace
↳ because of merge Sub-Routine

$T(n) = 2T(n/2) + \theta(n)$ [always]

$= \theta(n \log n)$

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

$$T(n) = 4T(n/2) + n \leq 4c(n/2)^3 + n$$

$$= (c/2)n^3 + n = \underbrace{cn^3 - ((c/2)n^3 - n)}_{\text{desired - residual}} \leq \underbrace{cn^3}_{\text{desired}}$$

Whenever $((c/2)n^3 - n) \geq 0$ if $c \geq 2, n \geq 1$

($(c/2)n^3 - n$: Residual)

4.3-1

Use the substitution method to show that each of the following recurrences defined on the reals has the asymptotic solution specified:

a. $T(n) = T(n-1) + n$ has solution $T(n) = O(n^2)$.

b. $T(n) = T(n/2) + \Theta(1)$ has solution $T(n) = O(\lg n)$.

c. $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n \lg n)$.

d. $T(n) = 2T(n/2 + 17) + n$ has solution $T(n) = O(n \lg n)$.

e. $T(n) = 2T(n/3) + \Theta(n)$ has solution $T(n) = \Theta(n)$.

f. $T(n) = 4T(n/2) + \Theta(n)$ has solution $T(n) = \Theta(n^2)$.

4.3-2

The solution to the recurrence $T(n) = 4T(n/2) + n$ turns out to be $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \leq cn^2$ fails. Then show how to subtract a lower-order term to make a substitution proof work.

4.3-3

The recurrence $T(n) = 2T(n-1) + 1$ has the solution $T(n) = O(2^n)$. Show that a substitution proof fails with the assumption $T(n) \leq c2^n$, where $c > 0$ is constant. Then show how to subtract a lower-order term to make a substitution proof work.

Weekley Test

2023

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

The master theorem

The master method depends upon the following theorem.

Theorem 4.1 (Master theorem)
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n)\,, \tag{4.17}$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:[1]

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

[1] "Introduction to Algorithm – CLRS, 4TH ed", Page 102

Using the master method

To use the master method, you determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider the recurrence $T(n) = 9T(n/3) + n$. For this recurrence, we have $a = 9$ and $b = 3$, which implies that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = n = O(n^{2-\epsilon})$ for any constant $\epsilon \leq 1$, we can apply case 1 of the master theorem to conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider the recurrence $T(n) = T(2n/3) + 1$, which has $a = 1$ and $b = 3/2$, which means that the watershed function is $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies since $f(n) = 1 = \Theta(n^{\log_b a} \lg^0 n) = \Theta(1)$. The solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence $T(n) = 3T(n/4) + n \lg n$, we have $a = 3$ and $b = 4$, which means that $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = n \lg n = \Omega(n^{\log_4 3+\epsilon})$, where ϵ can be as large as approximately 0.2, case 3 applies as long as the regularity condition holds for $f(n)$. It does, because for sufficiently large n, we have that $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. By case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Next, let's look at the recurrence $T(n) = 2T(n/2) + n \lg n$, where we have $a = 2$, $b = 2$, and $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies since $f(n) = n \lg n = \Theta(n^{\log_b a} \lg^1 n)$. We conclude that the solution is $T(n) = \Theta(n \lg^2 n)$.

We can use the master method to solve the recurrences we saw in Sections 2.3.2, 4.1, and 4.2.

Recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41, characterizes the running time of merge sort. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies because $f(n) = \Theta(n)$, and the solution is $T(n) = \Theta(n \lg n)$.

[2]

Recurrence (4.9), $T(n) = 8T(n/2) + \Theta(1)$, on page 84, describes the running time of the simple recursive algorithm for matrix multiplication. We have $a = 8$ and $b = 2$, which means that the watershed function is $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than the driving function $f(n) = \Theta(1)$—indeed, we have $f(n) = O(n^{3-\epsilon})$ for any positive $\epsilon < 3$—case 1 applies. We conclude that $T(n) = \Theta(n^3)$.

Finally, recurrence (4.10), $T(n) = 7T(n/2) + \Theta(n^2)$, on page 87, arose from the analysis of Strassen's algorithm for matrix multiplication. For this recurrence, we have $a = 7$ and $b = 2$, and the watershed function is $n^{\log_b a} = n^{\lg 7}$. Observing that $\lg 7 = 2.807355\ldots$, we can let $\epsilon = 0.8$ and bound the driving function $f(n) = \Theta(n^2) = O(n^{\lg 7-\epsilon})$. Case 1 applies with solution $T(n) = \Theta(n^{\lg 7})$.

[2] "Introduction to Algorithm – CLRS, 4TH ed", Page 104

When the master method doesn't apply

There are situations where you can't use the master theorem. For example, it can be that the watershed function and the driving function cannot be asymptotically compared. We might have that $f(n) \gg n^{\log_b a}$ for an infinite number of values of n but also that $f(n) \ll n^{\log_b a}$ for an infinite number of different values of n. As a practical matter, however, most of the driving functions that arise in the study of algorithms can be meaningfully compared with the watershed function. If you encounter a master recurrence for which that's not the case, you'll have to resort to substitution or other methods.

Even when the relative growths of the driving and watershed functions can be compared, the master theorem does not cover all the possibilities. There is a gap between cases 1 and 2 when $f(n) = o(n^{\log_b a})$, yet the watershed function does not grow polynomially faster than the driving function. Similarly, there is a gap between cases 2 and 3 when $f(n) = \omega(n^{\log_b a})$ and the driving function grows more than polylogarithmically faster than the watershed function, but it does not grow polynomially faster. If the driving function falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you'll need to use something other than the master method to solve the recurrence.

As an example of a driving function falling into a gap, consider the recurrence $T(n) = 2T(n/2) + n/\lg n$. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. The driving function is $n/\lg n = o(n)$, which means that it grows asymptotically more slowly than the watershed function n. But $n/\lg n$ grows only *logarithmically* slower than n, not *polynomially* slower. More precisely, equation (3.24) on page 67 says that $\lg n = o(n^\epsilon)$ for any constant $\epsilon > 0$, which means that $1/\lg n = \omega(n^{-\epsilon})$ and $n/\lg n = \omega(n^{1-\epsilon}) = \omega(n^{\log_b a - \epsilon})$. Thus no constant $\epsilon > 0$ exists such that $n/\lg n = O(n^{\log_b a - \epsilon})$, which is required for case 1 to apply. Case 2 fails to apply as well, since $n/\lg n = \Theta(n^{\log_b a} \lg^k n)$, where $k = -1$, but k must be nonnegative for case 2 to apply.[3]

To solve this kind of recurrence, you must use another method, such as the substitution method (Section 4.3) or the Akra-Bazzi method (Section 4.7). (Exercise 4.6-3 asks you to show that the answer is $\Theta(n \lg \lg n)$.) Although the master theorem doesn't handle this particular recurrence, it does handle the overwhelming majority of recurrences that tend to arise in practice.

[3] "Introduction to Algorithm – CLRS, 4TH ed", Page 105

Exercises

4.5-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1$.

b. $T(n) = 2T(n/4) + \sqrt{n}$.

c. $T(n) = 2T(n/4) + \sqrt{n}\lg^2 n$.

d. $T(n) = 2T(n/4) + n$.

e. $T(n) = 2T(n/4) + n^2$.

4.5-2

Professor Caesar wants to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into $n/4 \times n/4$ submatrices, and the divide and combine steps together will take $\Theta(n^2)$ time. Suppose that the professor's algorithm creates a recursive subproblems of size $n/4$. What is the largest integer value of a for which his algorithm could possibly run asymptotically faster than Strassen's?

4.5-3

Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-6 for a description of binary search.)

4.5-4

Consider the function $f(n) = \lg n$. Argue that although $f(n/2) < f(n)$, the regularity condition $af(n/b) \leq cf(n)$ with $a = 1$ and $b = 2$ does not hold for any constant $c < 1$. Argue further that for any $\epsilon > 0$, the condition in case 3 that $f(n) = \Omega(n^{\log_b a + \epsilon})$ does not hold. [4]

Read section 4.6 for better understanding.

[4] "Introduction to Algorithm – CLRS, 4TH ed", Page 106

Lecture 07: Divide & Conquer

Topics to be Covered:

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

Recall that for divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion ***bottoms out*** when it reaches a base case and the subproblem is small enough to solve directly without further recursing. [5]

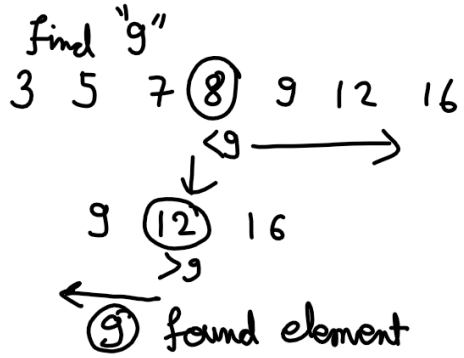
★ Binary Search

- Divide :- Check Middle Element
- Conquer :- Recursively search one sub array
- Combine :- Trivial (Not needed in case of Binary Search)
- Recurrence for Binary Search

$$\boldsymbol{T(n) = 1 * T\left(\frac{n}{2}\right) + \theta(n)}$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE II } (k = 0)$$

$$\boldsymbol{T(n) = \theta(\log n)}$$



[5] "Introduction to Algorithm – CLRS, 4TH ed", Page 76

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

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 of 2025

Week – 01

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

🔹 Insertion Sort Explained

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It's similar to how you might sort playing cards in your hands: you pick one card at a time and place it in the correct position among the already sorted cards.

🔧 How It Works (Step-by-Step)

1. Start from the second element (index 1), because a single element is already "sorted".
2. Compare the current element with the elements before it.
3. Shift the larger elements one position to the right.
4. Insert the current element into the correct position.
5. Repeat for all elements.

💡 Example

Let's sort the array:
`[5, 3, 4, 1, 2]`

Step-by-step:

Step	Array State	Explanation
1	`[5, 3, 4, 1, 2]`	Start from index 1 (`3`)
2	`[3, 5, 4, 1, 2]`	Insert 3 before 5
3	`[3, 4, 5, 1, 2]`	Insert 4 between 3 and 5
4	`[1, 3, 4, 5, 2]`	Insert 1 at the beginning
5	`[1, 2, 3, 4, 5]`	Insert 2 between 1 and 3

✅ Final Sorted Array: `[1, 2, 3, 4, 5]`

⏱️ Time Complexity

Case	Time Complexity
Best Case	O(n) (already sorted)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$ (reverse sorted)

```
# 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

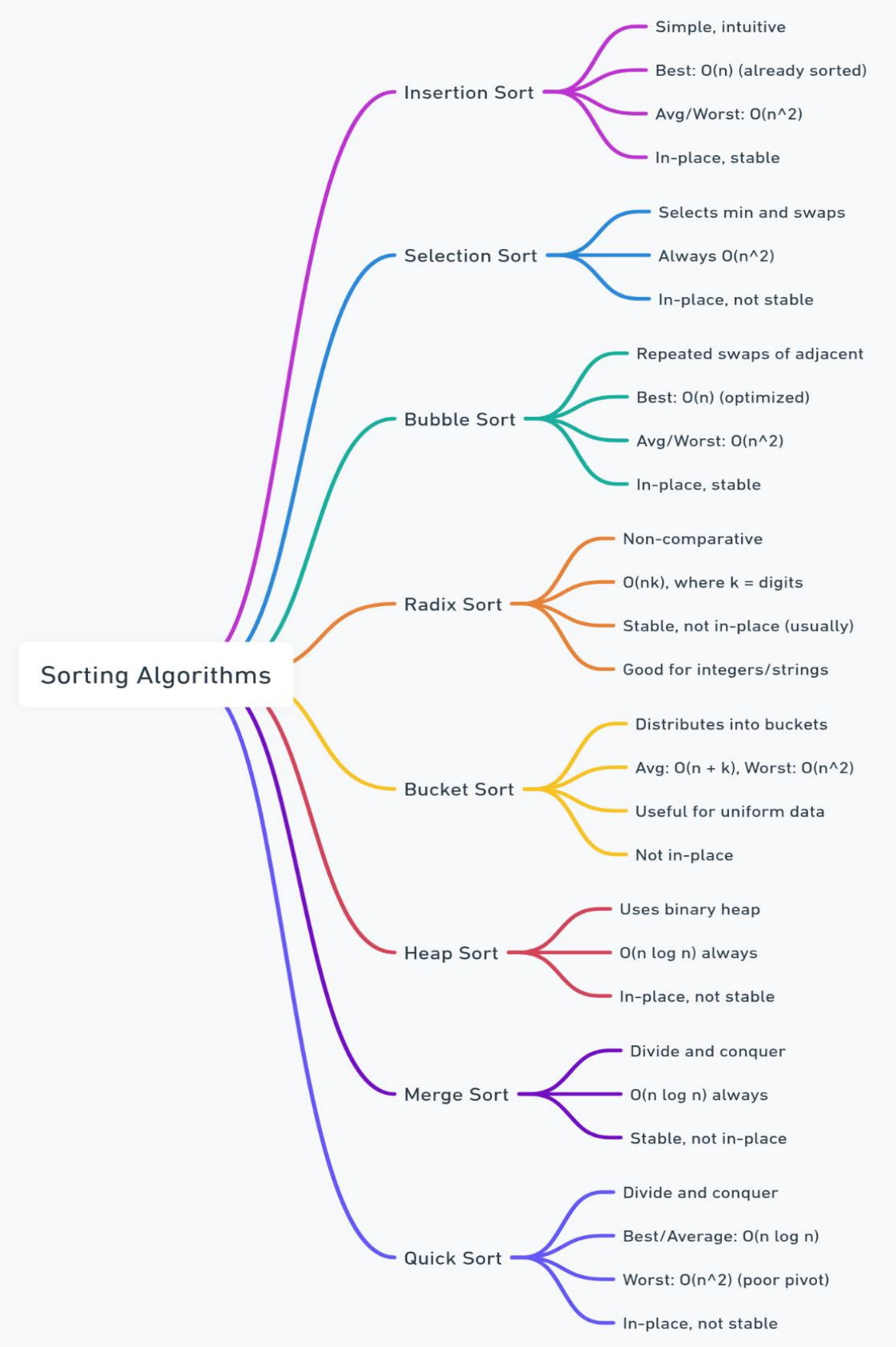
Add print() statements to see what's happening at each step:

```
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)
```

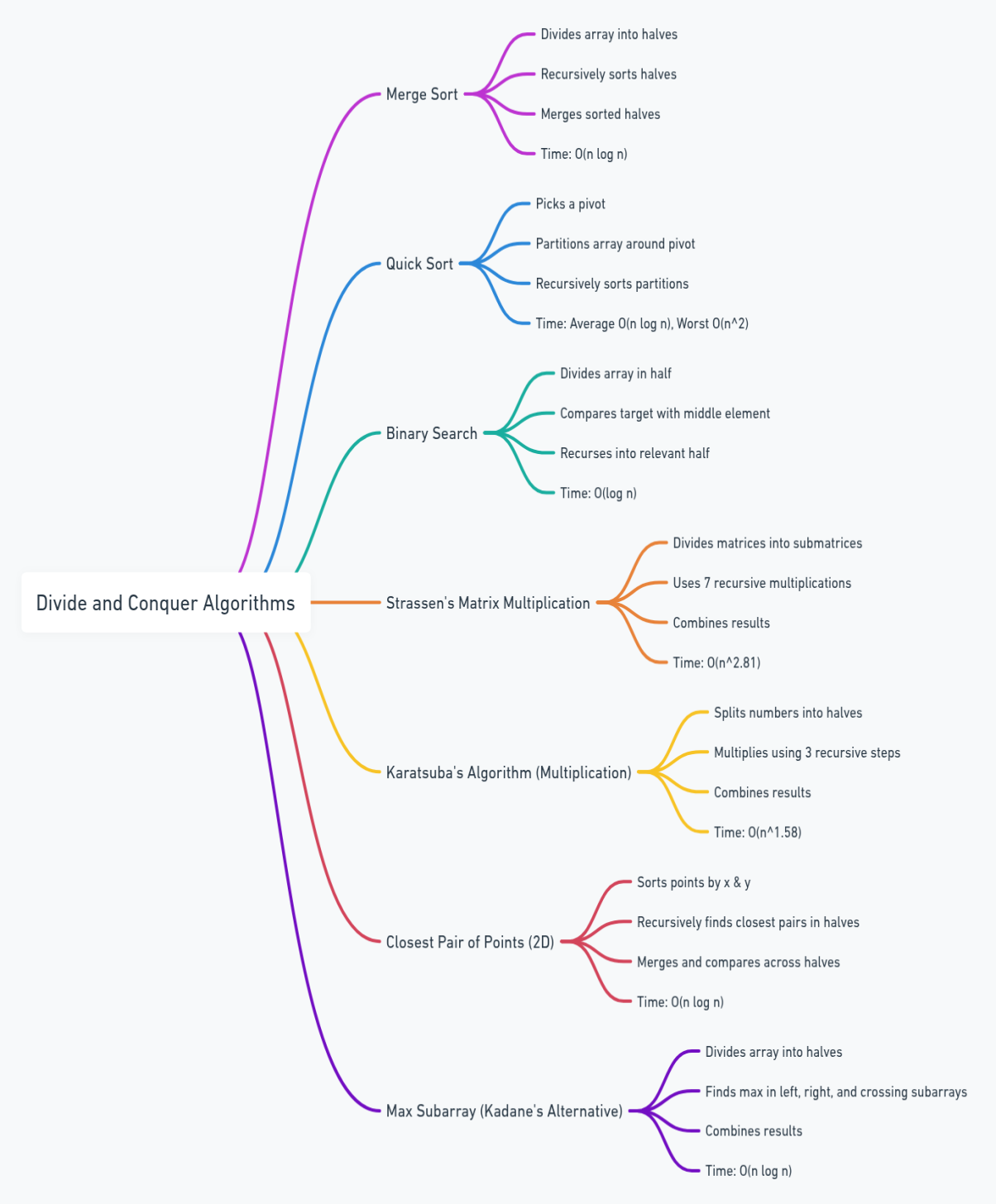
Create Insertion Sort in Decreasing order.

Whimsical Diagrams

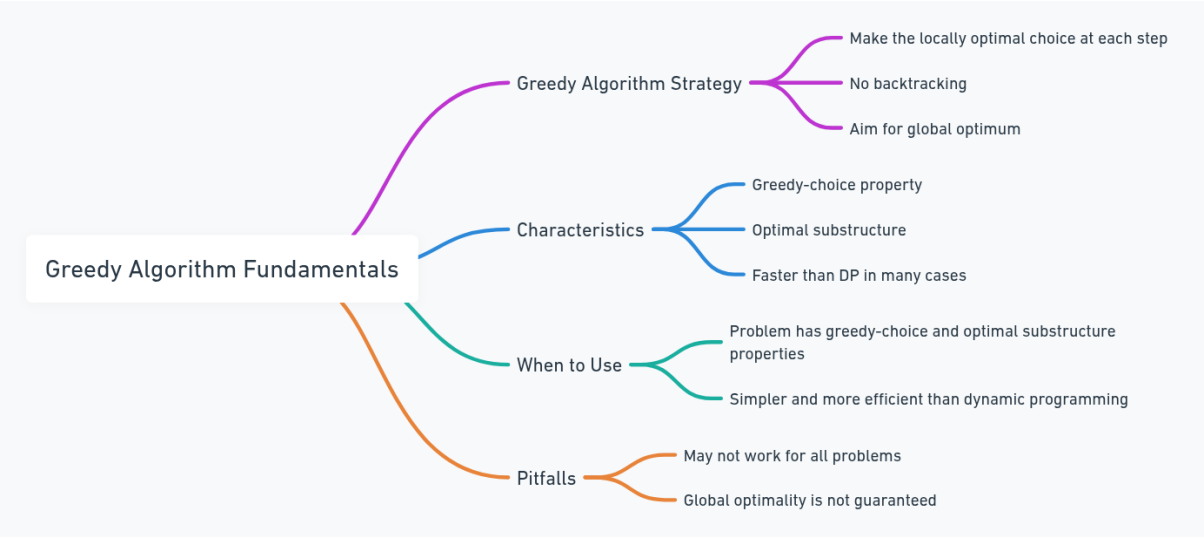
Sorting Technique

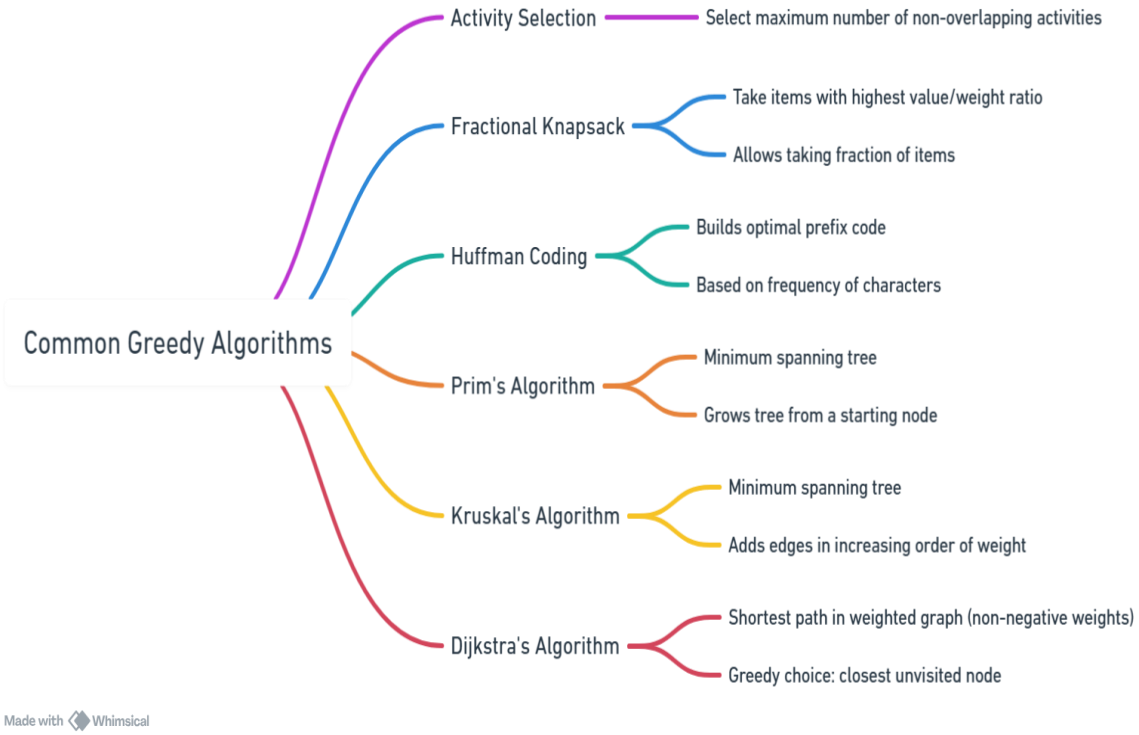


Divide and Conquer Algorithm

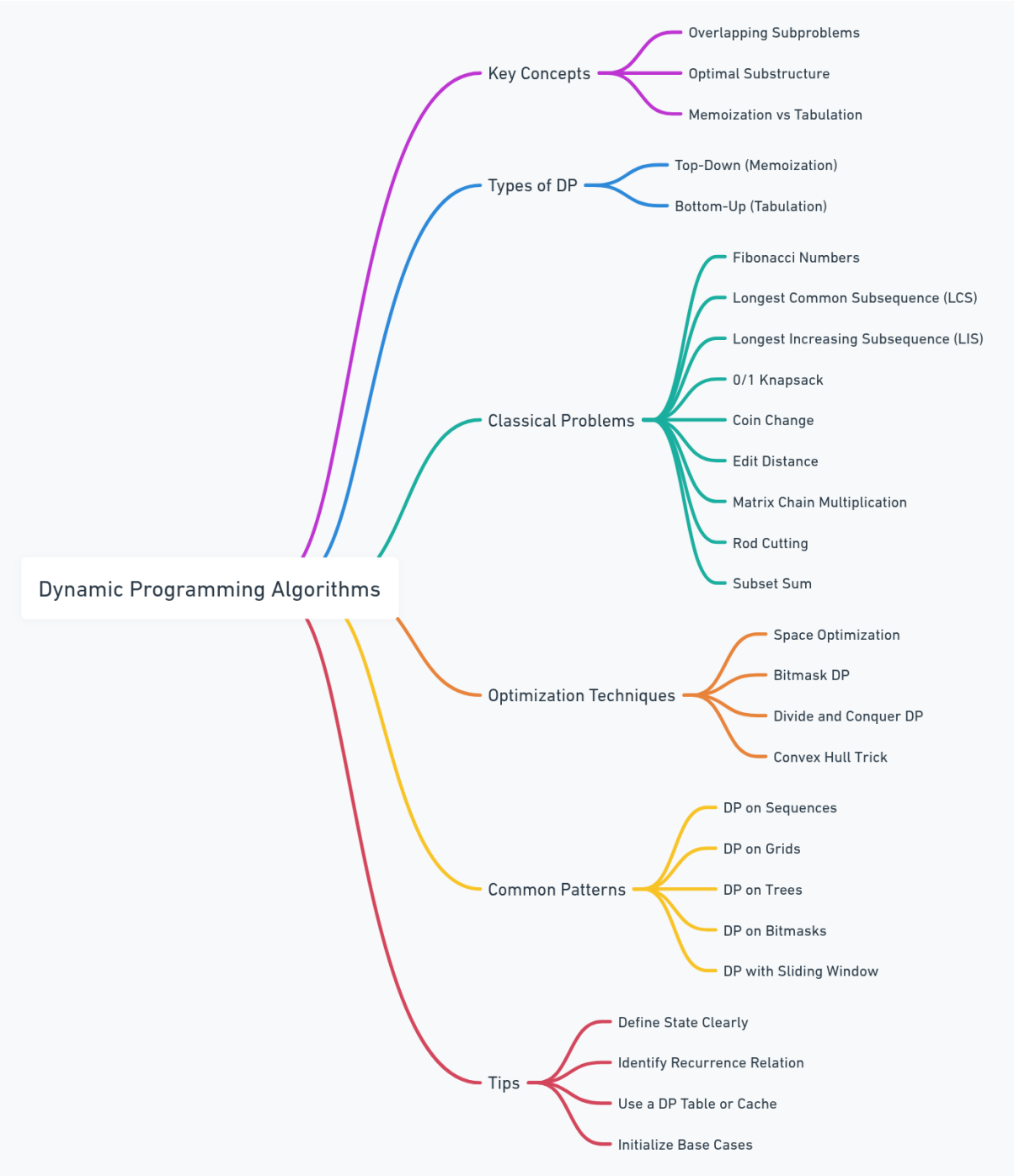


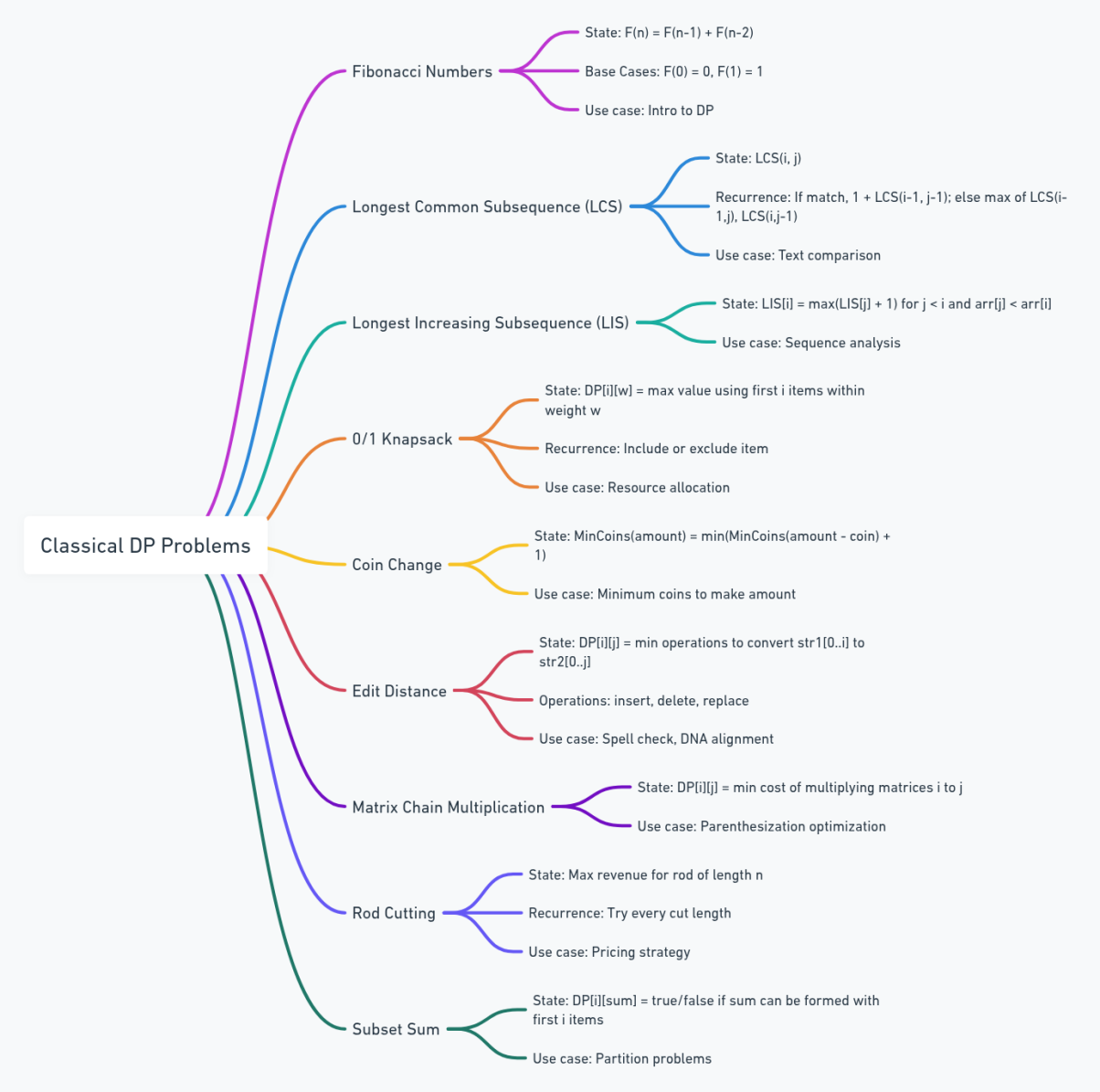
Greedy Algorithm



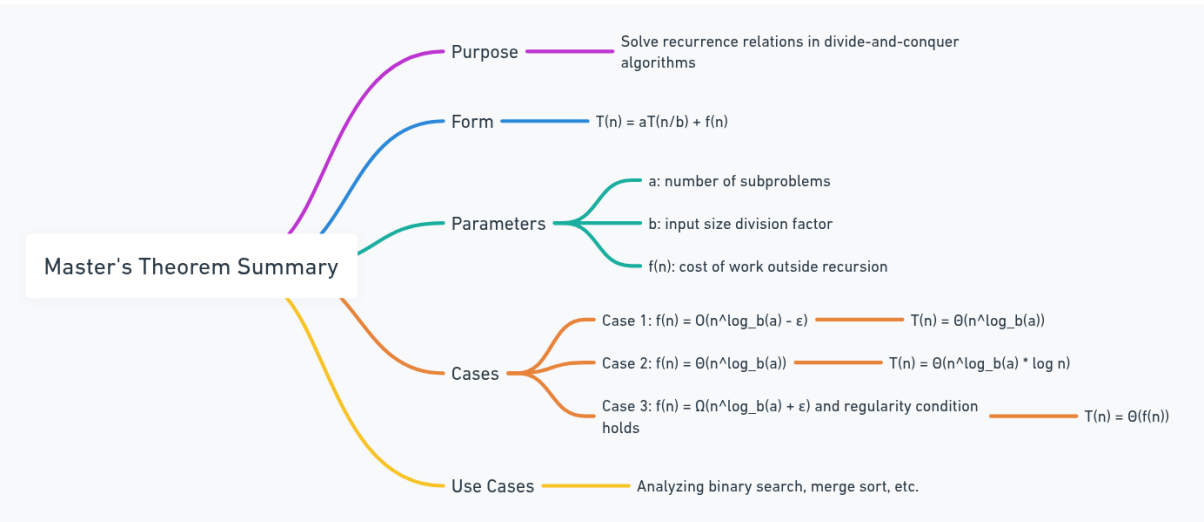


Dynamic Programming





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
Jupyter	Run notebooks for interactive coding
matplotlib	Visualization and plotting
numpy	Numerical computing
pandas	Data structures and manipulation
networkx	Graph theory & algorithm practice
rich	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
black	Auto-code formatter	`pip install black`
pytest	Testing algorithms	`pip install pytest`
ipython	Enhanced interactive shell	`pip install ipython`
pygraphviz	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

11- Implementation of KMP Algorithm

12- Implementation of Union by Rank Algorithm

13- Implementation of Various Sorting Algorithm

14- Implementation of Quick Sort Algorithm

15- Implementation of Merge Sort Algorithm

16- Implementation of Heap Sort Algorithm

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/)—a Python library for creating, manipulating, and visualizing complex networks—to help you **experiment interactively** with graph algorithms while learning them.

✅ Why NetworkX 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.

🏗️ 1. Creating Graphs

python Copy Edit

```python
import networkx as nx

# Undirected Graph
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'D'), ('A', 'D')])

# Directed Graph
DG = nx.DiGraph()
DG.add_weighted_edges_from([('A', 'B', 5), ('B', 'C', 2), ('A', 'C', 9)])
```

You can also use `MultiGraph()` or `MultiDiGraph()` for multigraphs (with parallel edges).

🔄 2. Breadth-First Search (BFS)

python Copy Edit

```python
import networkx as nx

G = nx.Graph()
G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 4)])

# BFS traversal from node 0
bfs_edges = list(nx.bfs_edges(G, source=0))
bfs_nodes = list(nx.bfs_tree(G, source=0))

print("BFS traversal:", bfs_edges)
```

🔍 Use:

Explore all reachable nodes from a starting point in layers.

🧬 3. Depth-First Search (DFS)

python Copy Edit

```python
dfs_edges = list(nx.dfs_edges(G, source=0))
print("DFS traversal:", dfs_edges)
```

🔍 Use:

Used in solving puzzles, topological sorting, cycle detection, etc.

📐 4. Dijkstra's Shortest Path

python Copy Edit

```python
DG = nx.DiGraph()
DG.add_weighted_edges_from([
    ('A', 'B', 2),
    ('A', 'C', 5),
    ('B', 'C', 1),
    ('C', 'D', 3)
])

path = nx.dijkstra_path(DG, source='A', target='D')
length = nx.dijkstra_path_length(DG, source='A', target='D')

print("Shortest path:", path)
print("Path length:", length)
```

🔍 Use:

Finds the shortest path in weighted graphs with non-negative weights.

⛓️ 5. Cycle Detection

python Copy Edit

```python
try:
    cycle = nx.find_cycle(G)
    print("Cycle found:", cycle)
except nx.exception.NetworkXNoCycle:
    print("No cycle found")
```

🔍 Use:

Useful in detecting infinite loops or deadlocks.

📊 6. Topological Sorting

python Copy Edit

```python
DAG = nx.DiGraph()
DAG.add_edges_from([('cook', 'eat'), ('shop', 'cook'), ('study', 'pass')])

order = list(nx.topological_sort(DAG))
print("Topological Order:", order)
```

⚠️ Note:

Only works on Directed Acyclic Graphs (DAGs).

🌐 7. Graph Visualization

python Copy Edit

```python
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_weighted_edges_from([('A', 'B', 4), ('A', 'C', 1), ('C', 'D', 2)])

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000, font_size=14)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.show()
```

🔍 Use:

Visual intuition is critical for understanding how graphs behave.

📁 Summary Table

Task	Function(s)
Create Graph	`Graph()`, `DiGraph()`, `add_edge()`
BFS	`bfs_edges()`, `bfs_tree()`
DFS	`dfs_edges()`
Shortest Path	`dijkstra_path()`, `dijkstra_path_length()`
Cycle Detection	`find_cycle()`
Topological Sorting	`topological_sort()`
Visualization	`draw()`, `draw_networkx_edge_labels()`

📚 Other Use Cases:

Algorithm / Problem	`networkx` Feature Used
Cycle detection	`nx.find_cycle()`
DFS/BFS	`nx.dfs_edges()` / `nx.bfs_edges()`
PageRank	`nx.pagerank()`
Topological Sort	`nx.topological_sort()`
Graph Coloring	Custom coloring or external libs
Minimum Spanning Tree	`nx.minimum_spanning_tree()`

🧪 Example Use Case: **Shortest Path using Dijkstra's Algorithm**

Let's say you want to compute the shortest path between two cities on a road map.

```
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)