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

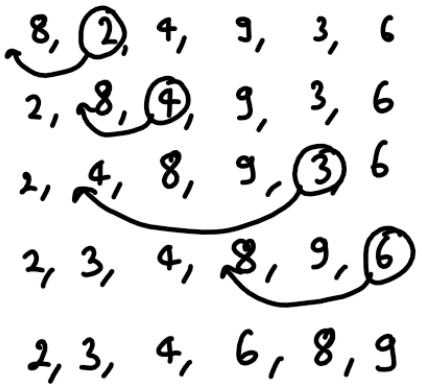
Input: 9 3 5 0 4 7
Output: 0 3 4 5 7 9

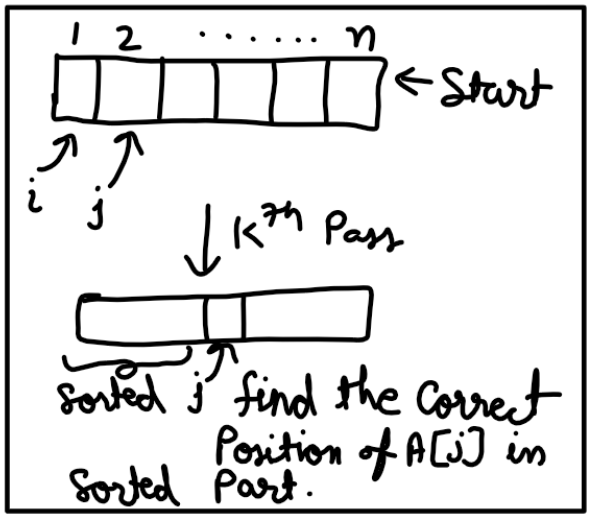
Pseudo Code:- Insertion Sort

```
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 $f(n) \le c * g(n)\ \forall\ n \ge n_0\}$

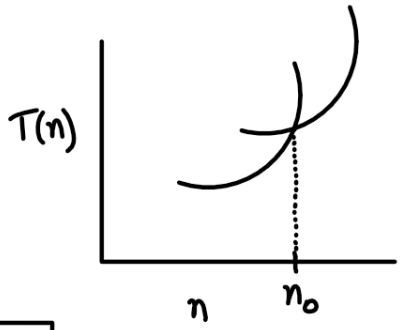
★ Ω Notation

$\Omega(g(n)) = \{f(n)$: ∃ 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

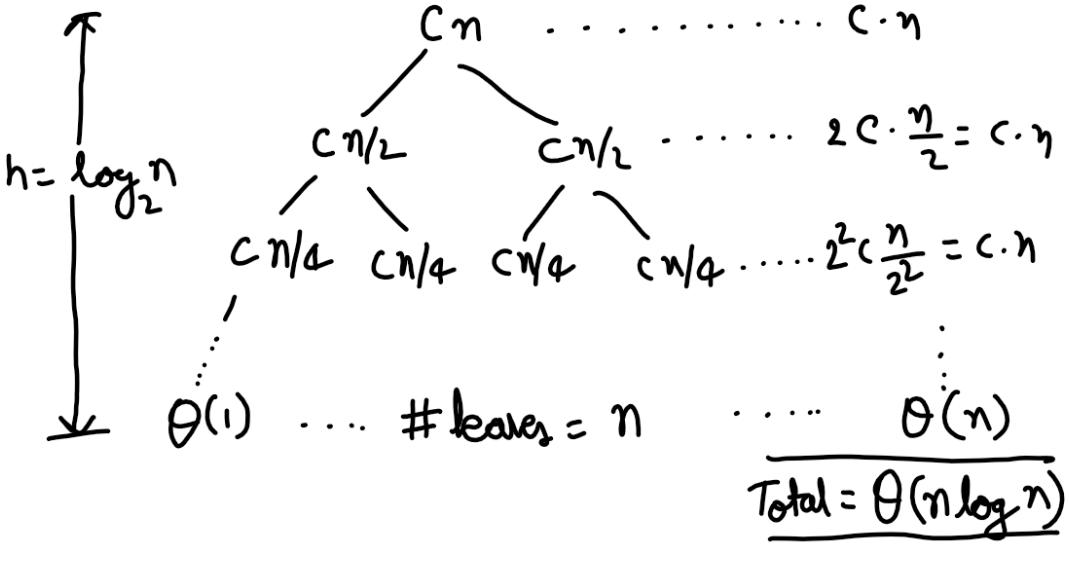
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$$

★ Recurrence for Merge Sort

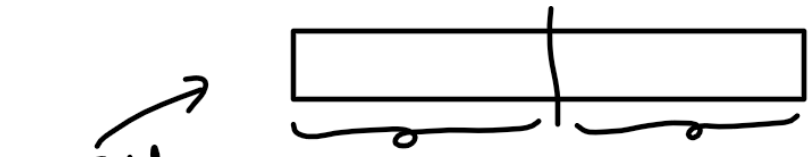
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

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.

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]

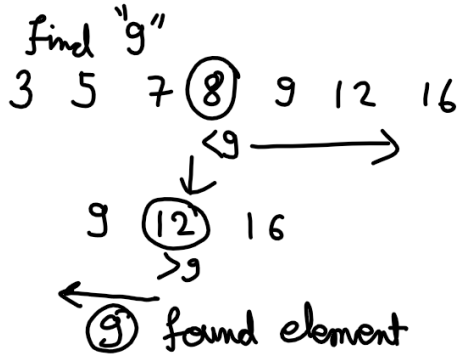
★ Binary Search

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

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

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

$$T(n) = \theta(\log n)$$



★ Powering a Number

- Problem:- Compute a^n, $n \in N$
- Naive Algorithm:- $\theta(n)$
- Divide and Conquer Algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

$$\Rightarrow T(n) = T(n/2) + \theta(1)$$

$$\Rightarrow T(n) = \theta(\log_2 n)$$

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

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

Topics to be Covered:

- Fibonacci Number
- Strassen's Matrix Multiplication Algorithm

★ Fibonacci Number

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}, & n \geq 2 \end{cases}$$

Q. Find n^{th} Fibonacci Number?

$f_n = \frac{\phi^n}{\sqrt{5}}$ where $\phi = \frac{1+\sqrt{5}}{2}$ → Golden Ratio

Naive Recursive Squaring	Bottom - Up
$f_n = \phi^n/\sqrt{n}$] Recursive Squaring $\theta(\lg n)$ times!	* Keep on computing $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two Previous.
* Unreliable, since floating-Point arithmetic is prone to round off errors.	* $\theta(n)$

★ Recursive Squaring

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Time = $\theta(\lg n)$

$$f_{2017} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2017}$$

Desired Result = $\begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix}$

★ Matrix Multiplication

- Input :- A = $[a_{ij}]$, B = $[b_{ij}]$
- Output :- C = $[c_{ij}]$, where i, j = 1, 2, 3, ... , n
- $C_{i,j} = \sum_{k=1}^{n} a_{ix} * b_{xj}$, Where i, j = 1, 2, 3, ..., n

Standard Algorithm:-

1.	For $i \leftarrow 1$ to n
2.	do for j $\leftarrow 1$ *to* n
3.	do $c_{ij} \leftarrow 0$
4.	for k $\leftarrow$1 to n
5.	do $c_{ij} \leftarrow c_{ij} + a_{ix} * b_{xj}$

Time Complexity = $\theta(n^3)$

Lecture 09: Strassen's Algorithm

Topics to be Covered:

- Matrix Multiplication using Divide & Conquer
- Strassen's Idea

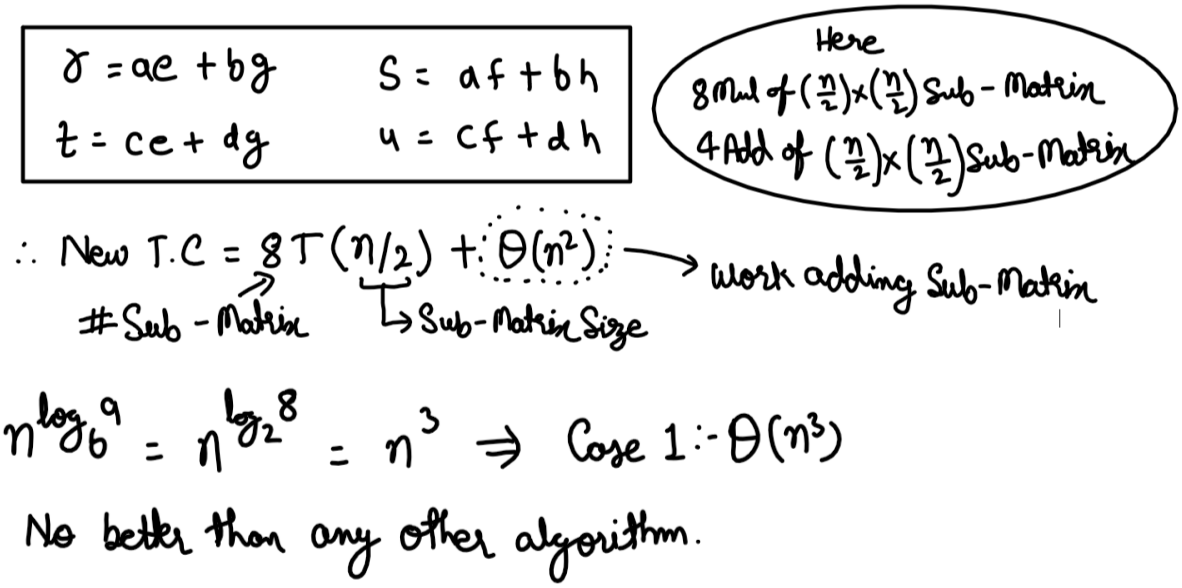
$$A * B = [a_{ij}]_{n \times n} [b_{ij}]_{n \times n} = C = [c_{ij}]_{n \times n} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \Big] \underbrace{\theta(n^3)}_{\text{Standard}}$$

Apply Divide & Conquer Technique

$n \times n$ Matrix $\longrightarrow$ 2×2 matrix of $(n/2) \times (n/2)$ Sub-Matrix

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} \longrightarrow \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C \longrightarrow A \cdot B$$



✅ Strassen's Matrix Multiplication Algorithm [6]— Explained Simply

Strassen's algorithm is a **faster matrix multiplication** method than the standard $\theta(n^3)$ 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$n * n$:

- Normally, to multiply, we do n^3 scalar multiplications.
- Strassen reduces this to approximately $O(n^{2.81})$ 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 $n * n$, 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 $\left(\frac{n}{2} * \frac{n}{2}\right)$

[6] https://chatgpt.com/c/685143fb-437c-800c-952c-551b01ec216b

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 $\left(\frac{n}{2} * \frac{n}{2}\right)$

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 $C11, C12, C21, C22$ into full matrix C.

🔍 **Explanation of Efficiency**

- **Standard method**: n^3 multiplications
- **Strassen**: Reduces multiplication count from 8 to 7
- **Time Complexity**: $T(n) = 7T(n/2) + O(n^2)$
    - Solving this gives:

$$T(n) = O(n^{\log 2.7}) \approx O(n^{2.81})$$

📝 **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.

❗ Strassen only works for **powers of 2** (like 2×2, 4×4, etc.)

Since 3 is **not** a power of 2, we must **pad the matrix to 4×4** by adding zero rows/columns.

🧮 Let's define two 3×3 matrices:

Let

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

🔄 Step 1: Pad to 4×4 matrices

$$A' = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 5 & 5 & 6 & 0 \\ 4 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{ and } B' = \begin{bmatrix} 9 & 8 & 7 & 0 \\ 6 & 5 & 4 & 0 \\ 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

🔀 Step 2: Split into 2×2 blocks

Each matrix is split into 4 blocks:

$$A' = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B' = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Where:

- $A_{11} = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$
- $A_{12} = \begin{bmatrix} 3 & 0 \\ 6 & 0 \end{bmatrix}$
- $A_{21} = \begin{bmatrix} 7 & 8 \\ 0 & 0 \end{bmatrix}$
- $A_{22} = \begin{bmatrix} 9 & 0 \\ 0 & 0 \end{bmatrix}$

Same for **B'**:

- $B_{11} = \begin{bmatrix} 9 & 8 \\ 6 & 5 \end{bmatrix}$
- $B_{12} = \begin{bmatrix} 7 & 0 \\ 4 & 0 \end{bmatrix}$
- $B_{21} = \begin{bmatrix} 3 & 2 \\ 0 & 0 \end{bmatrix}$
- $B_{22} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

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

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

M1 = (A11 + A22) × (B11 + B22)

- $A11 + A22 = \begin{bmatrix} 1+9 & 2+0 \\ 4+0 & 5+0 \end{bmatrix} = \begin{bmatrix} 10 & 2 \\ 4 & 5 \end{bmatrix}$
- $B11 + B22 = \begin{bmatrix} 9+1 & 8+0 \\ 6+0 & 5+0 \end{bmatrix} = \begin{bmatrix} 10 & 8 \\ 6 & 5 \end{bmatrix}$

Now multiply:

$$M1 = \begin{bmatrix} 10 & 2 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 10 & 8 \\ 6 & 5 \end{bmatrix}$$

$$M1 = \begin{bmatrix} (10 \times 10 + 2 \times 6) & (10 \times 8 + 2 \times 5) \\ (4 \times 10 + 5 \times 6) & (4 \times 8 + 5 \times 5) \end{bmatrix} = \begin{bmatrix} 100 + 12 & 80 + 10 \\ 40 + 30 & 32 + 25 \end{bmatrix} = \begin{bmatrix} 112 & 90 \\ 70 & 57 \end{bmatrix}$$

✅ **M2 = (A21 + A22) × B11**

$$A21 + A22 = \begin{bmatrix} 16 & 8 \\ 0 & 0 \end{bmatrix}$$

$$M2 = \begin{bmatrix} 16 \times 9 + 8 \times 6 & 16 \times 8 + 8 \times 5 \\ 0 \times 9 + 0 \times 6 & 0 \times 8 + 0 \times 5 \end{bmatrix} = \begin{bmatrix} 192 & 168 \\ 0 & 0 \end{bmatrix}$$

✅ **M3 = A11 × (B12 − B22)**

$$B12 - B22 = \begin{bmatrix} 6 & 0 \\ 4 & 0 \end{bmatrix}$$

$$M3 = \begin{bmatrix} 1 \times 6 + 2 \times 4 & 1 \times 0 + 2 \times 0 \\ 4 \times 6 + 5 \times 4 & 4 \times 0 + 5 \times 0 \end{bmatrix} = \begin{bmatrix} 14 & 0 \\ 44 & 0 \end{bmatrix}$$

✅ **M4 = A22 × (B21 − B11)**

$$B21 - B11 = \begin{bmatrix} -6 & -6 \\ -6 & -5 \end{bmatrix}$$

$$M4 = \begin{bmatrix} 9 \times -6 + 0 \times -6 & 9 \times -6 + 0 \times -5 \\ 0 \times -6 + 0 \times -6 & 0 \times -6 + 0 \times -5 \end{bmatrix} = \begin{bmatrix} -54 & -54 \\ 0 & 0 \end{bmatrix}$$

✅ **M5 = (A11 + A12) × B22**

$$A11 + A12 = \begin{bmatrix} 4 & 2 \\ 10 & 5 \end{bmatrix}$$

$$M5 = \begin{bmatrix} 4 \times 1 + 2 \times 0 & 4 \times 0 + 2 \times 0 \\ 10 \times 1 + 5 \times 0 & 10 \times 0 + 5 \times 0 \end{bmatrix} = \begin{bmatrix} 4 & 0 \\ 10 & 0 \end{bmatrix}$$

✅ M6 = (A21 − A11) × (B11 + B12)

$$A21 - A11 = \begin{bmatrix} 6 & 6 \\ -4 & -5 \end{bmatrix}, \quad B11 + B12 = \begin{bmatrix} 16 & 8 \\ 10 & 5 \end{bmatrix}$$

$$M6 = \begin{bmatrix} 6 \times 16 + 6 \times 10 & 6 \times 8 + 6 \times 5 \\ -4 \times 16 + -5 \times 10 & -4 \times 8 + -5 \times 5 \end{bmatrix} = \begin{bmatrix} 156 & 78 \\ -114 & -57 \end{bmatrix}$$

✅ M7 = (A12 − A22) × (B21 + B22)

$$A12 - A22 = \begin{bmatrix} -6 & 0 \\ 6 & 0 \end{bmatrix}, \quad B21 + B22 = \begin{bmatrix} 4 & 2 \\ 0 & 0 \end{bmatrix}$$

$$M7 = \begin{bmatrix} -6 \times 4 + 0 \times 0 & -6 \times 2 + 0 \times 0 \\ 6 \times 4 + 0 \times 0 & 6 \times 2 + 0 \times 0 \end{bmatrix} = \begin{bmatrix} -24 & -12 \\ 24 & 12 \end{bmatrix}$$

🧩 Step 4: Reconstruct 4x4 result from blocks

Using:

$$\begin{aligned} C11 &= M1 + M4 - M5 + M7 \\ C12 &= M3 + M5 \\ C21 &= M2 + M4 \\ C22 &= M1 - M2 + M3 + M6 \end{aligned}$$

🔷 C11:

$$= \begin{bmatrix} 112 & 90 \\ 70 & 57 \end{bmatrix} + \begin{bmatrix} -54 & -54 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 4 & 0 \\ 10 & 0 \end{bmatrix} + \begin{bmatrix} -24 & -12 \\ 24 & 12 \end{bmatrix} = \begin{bmatrix} 30 & 24 \\ 84 & 69 \end{bmatrix}$$

🔷 C12:

$$= \begin{bmatrix} 14 & 0 \\ 44 & 0 \end{bmatrix} + \begin{bmatrix} 4 & 0 \\ 10 & 0 \end{bmatrix} = \begin{bmatrix} 18 & 0 \\ 54 & 0 \end{bmatrix}$$

🔷 C21:

$$= \begin{bmatrix} 192 & 168 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} -54 & -54 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 138 & 114 \\ 0 & 0 \end{bmatrix}$$

🔷 C22:

$$= \begin{bmatrix} 112 & 90 \\ 70 & 57 \end{bmatrix} - \begin{bmatrix} 192 & 168 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 14 & 0 \\ 44 & 0 \end{bmatrix} + \begin{bmatrix} 156 & 78 \\ -114 & -57 \end{bmatrix} = \begin{bmatrix} 90 & 0 \\ 0 & 0 \end{bmatrix}$$

🧮 Final Step: Combine 2×2 blocks → remove padding → 3×3

Final matrix:

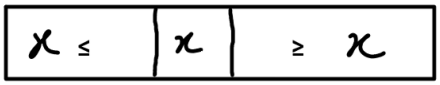
$$C = A \times B = \begin{bmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{bmatrix}$$

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



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

Key: - Linear Time Partition Sub-Routine

Partitioning Subroutine

```
1.  PARTITION (A, p, q)
2.      x ⟵ A[p]                          // A[p] = Pivot
3.      i ⟵p
4.      for j ⟵p + 1 to q
5.          do if A[j] ≤ x
6.              then i ⟵ i + 1
7.                   exchange A[i] ⟷ A[j]
8.      exchange A[p] ⟷ A[i]
9.      return i
```

Example of Partitioning

⑥	10	13	5	8	3	2	11
⑥	5	13	10	8	3	2	11
⑥	5	3	10	8	13	2	11
⑥	5	3	2	8	13	10	11
2	5	3	⑥	8	13	10	11

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

Q. 1. What operation does the Insertion Sort use to move numbers from the unsorted section to the sorted section of the list?

(a) Finding the minimum value
(b) Swapping
(c) Finding out an pivot value
(d) None of the above

b [7]

Q 2. If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$, then $f(n) = g(n)$. This statement is

(a) True
(b) False

b

Q 3. Consider that we apply insertion sort to sort the array $[25, 17, 31, 13, 2]$. What will be the order of the elements after 3^{rd} iteration?

(a) $13, 2, 17, 25, 31$
(b) $13, 17, 25, 31, 2$
(c) $17, 25, 31, 13, 2$
(d) None of these

b

Q 4. Suppose we have the following list of numbers to sort: $[15, 5, 4, 18, 12, 19, 14, 10, 8, 20]$ which list represents the partially sorted list after three complete passes of insertion sort?

(a) $[4, 5, 12, 15, 14, 10, 8, 18, 19, 20]$
(b) $[15, 5, 4, 10, 12, 8, 14, 18, 19, 20]$
(c) $[4, 5, 15, 18, 12, 19, 14, 10, 8, 20]$
(d) $[15, 5, 4, 18, 12, 19, 14, 8, 10, 20]$

c

Q 5. $\frac{1}{2}n^2 + 3n$ is

(a) $\theta(\log n)$

(b) $\theta(n)$

(c) $\theta(n^2)$

(d) $\theta(n \log n)$

(c) is the correct answer

[7] https://onlinecourses.nptel.ac.in/noc23_cs88/unit?unit=17&assessment=154

Q 6. If both f and h are individually $\mathcal{O}(n)$, then

(a) $f(n) \pm h(n)$ are $\mathcal{O}(n)$
(b) $f(n) + h(n)$ is $\mathcal{O}(n)$ but $f(n) - h(n)$ is not $\mathcal{O}(n)$
(c) $f(n) - h(n)$ is $\mathcal{O}(n)$ but $f(n) + h(n)$ is not $\mathcal{O}(n)$ a
(d) The given information is incomplete to make any conclusions.

Q 7. The space complexity of Merge sort is

a. $O(\log n)$
b. $O(\log \log n)$
c. $O(n)$
d. $O(n \log n)$ (c) is the correct answer

Q 8. Merge sort uses

a. Divide and Conquer strategy
b. Backtracking Strategy
c. Heuristic Search
d. 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?

a. $O(n^2)$
b. $O(n)$
c. $O(n^3)$
d. $O(\log n)$

Q 10. Consider the following

(*i*) $\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$.
(*ii*) The worst case time complexity for merge sort is $\mathcal{O}(\log n)$.

Select the correct option from below.

(a) (*i*) and (*ii*) both are true.
(b) Only (*ii*) is true.
(c) Only (*i*) is true. c
(d) (*i*) and (*ii*) both are false.

2025

Week – 02

2023

Q 1. To ……, Master's Theorem is used.

a. Solving Recurrence
b. Solving Iterative relation
c. Analysing loops
d. None of the above (a) is the correct answer

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

a. $O(\sqrt{n})$
b. $O(\log n)$
c. $O(n^3)$
d. $O(2^n)$ (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]

a. 8
b. 7
c. 6
d. 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)

a. $P_1 + P_7$
b. $P_1 + P_3$
c. $P_2 + P_4 – P_5 + P_7$
d. $P_3 + P_4$ (d) is the correct answer

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

$$T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$$

a. $T(n) = \theta(n^{0.51})$
b. $T(n) = \theta(n)$
c. $T(n) = \theta(log\, n)$
d. $T(n) = \theta(n^2)$ (a) is the correct answer

Q 6. Multiplying a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a sub routine produce

a. k multiplication of $n \times n$ matrices
b. k^2 multiplication of $n \times n$ matrices
c. k multiplication of $k \times n$ matrices
d. k^2 multiplication of $n \times k$ 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?

a. [9, 3, 10, 13, 12]
b. [9, 3, 10, 13, 12, 14]
c. [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
d. [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

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

a. Yes
b. No (b) is the correct answer

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

a. $O(n^{2.81})$
b. $O(n^{2.11})$
c. $O(n^{1.81})$
d. $O(n^{0.81})$ (a) is the correct answer

Q 10. Let $\mathbf{r_n}$ be the number of n-bit strings that do NOT contain two consecutive 1's. Which of the following is the recurrence relation for $\mathbf{r_n}$?

a. $\mathbf{r_n} = \mathbf{r_{n-1}} + 2\ \mathbf{r_{n-2}}$
b. $\mathbf{r_n} = \mathbf{r_{n-1}} + \mathbf{r_{n-2}}$
c. $\mathbf{r_n} = 2\mathbf{r_{n-1}} + \mathbf{r_{n-2}}$
d. $\mathbf{r_n} = \mathbf{r_{n-1}} + 2\ \mathbf{r_{n-2}}$

(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

🔹 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

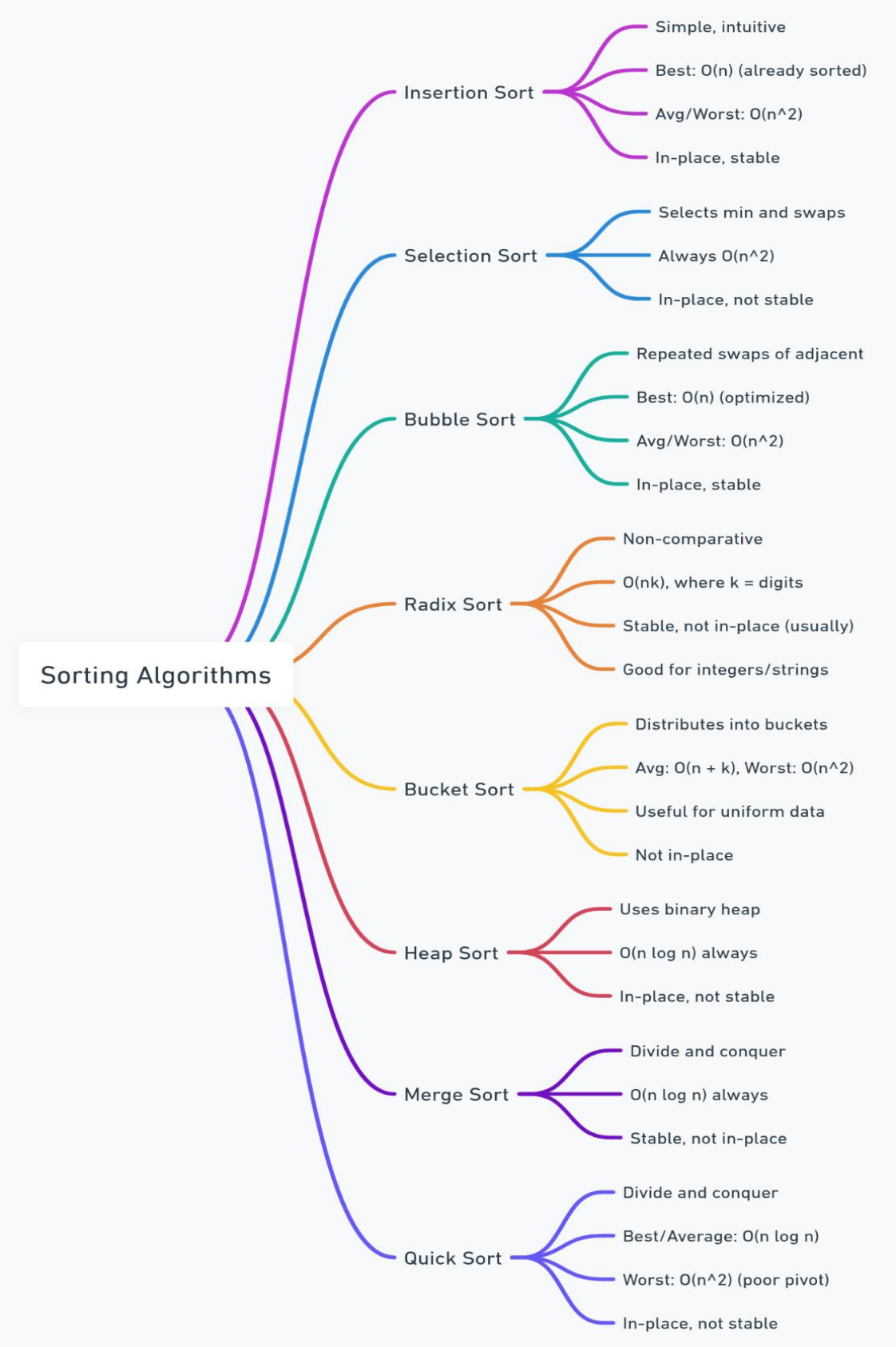
Add print() statements to see what's happening at each step:

```
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)
```

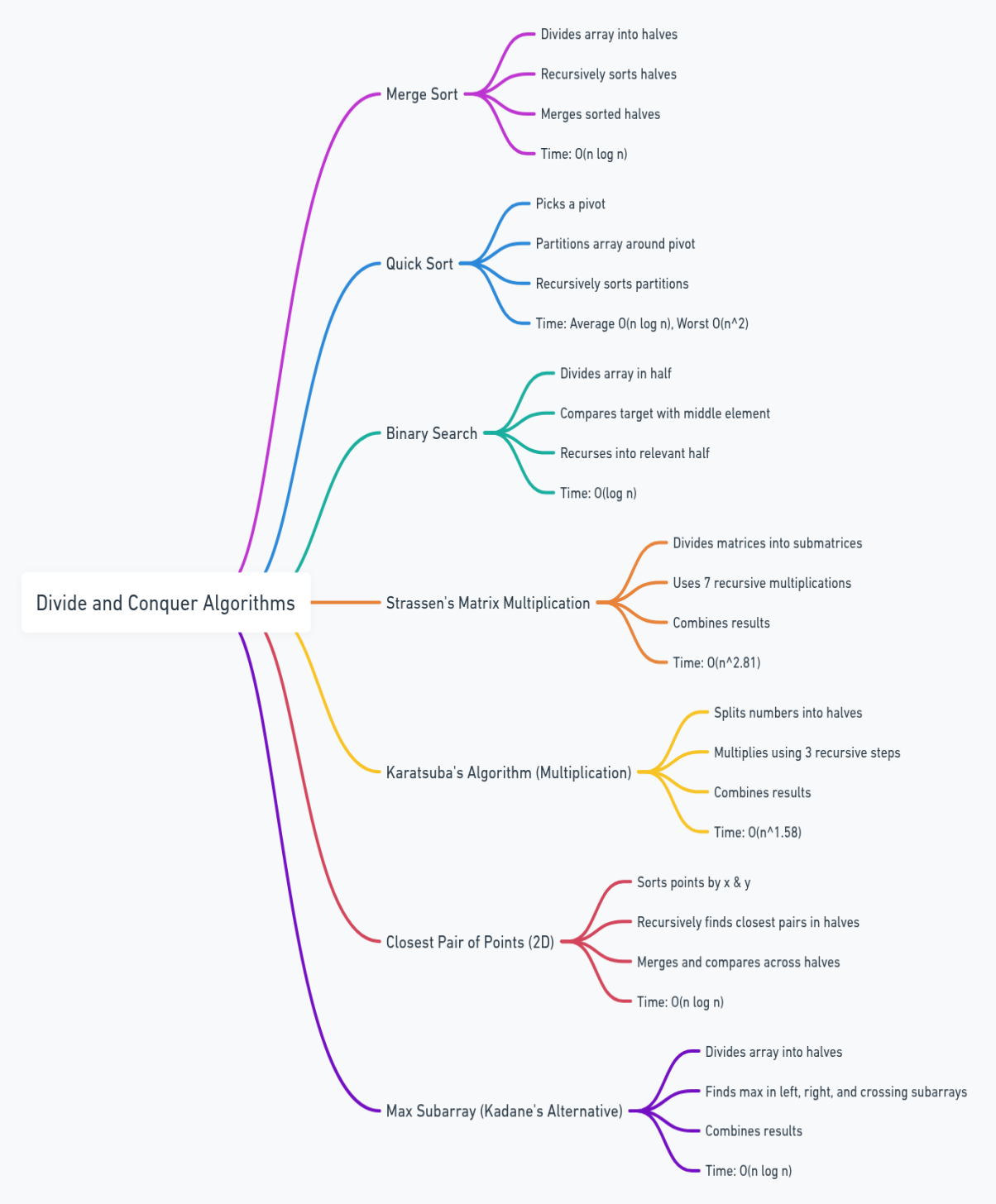
Create Insertion Sort in Decreasing order.

Whimsical Diagrams

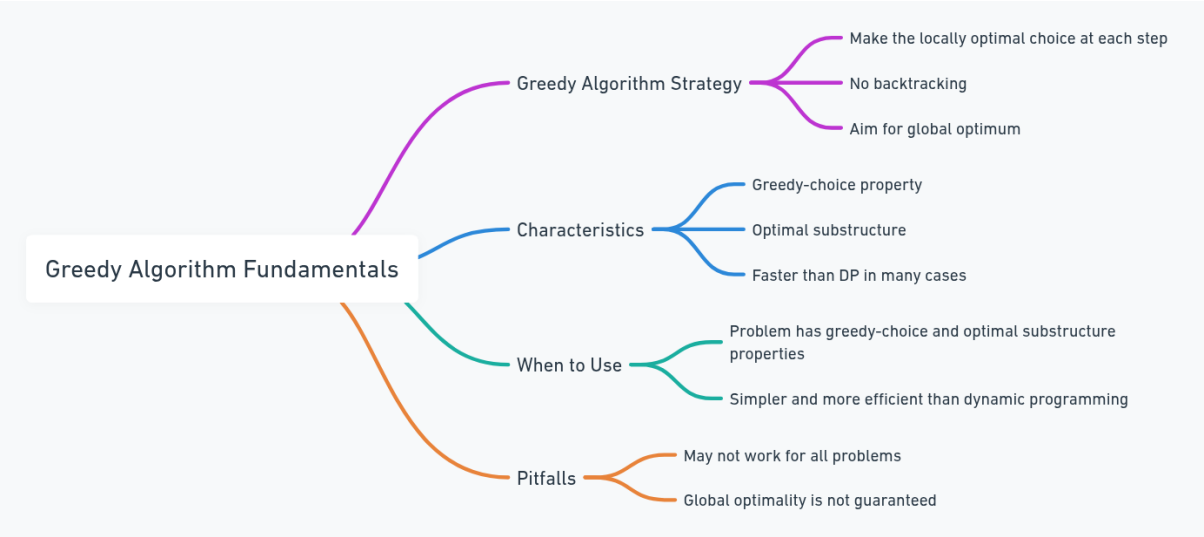
Sorting Technique

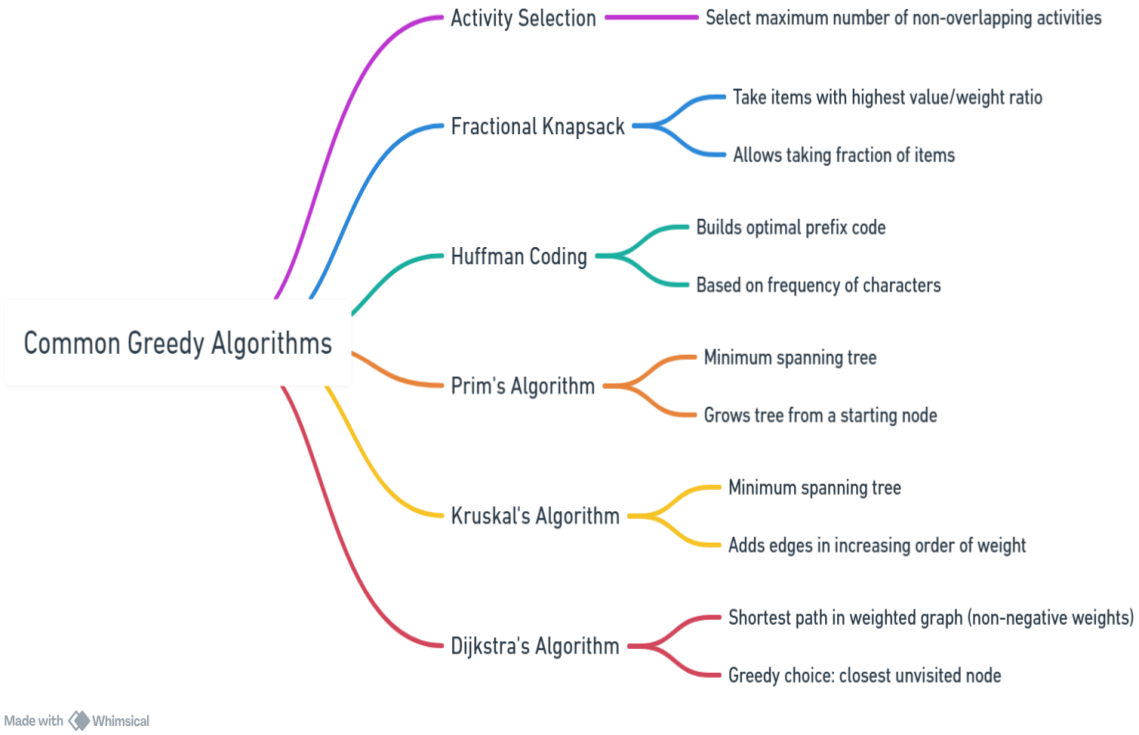


Divide and Conquer Algorithm

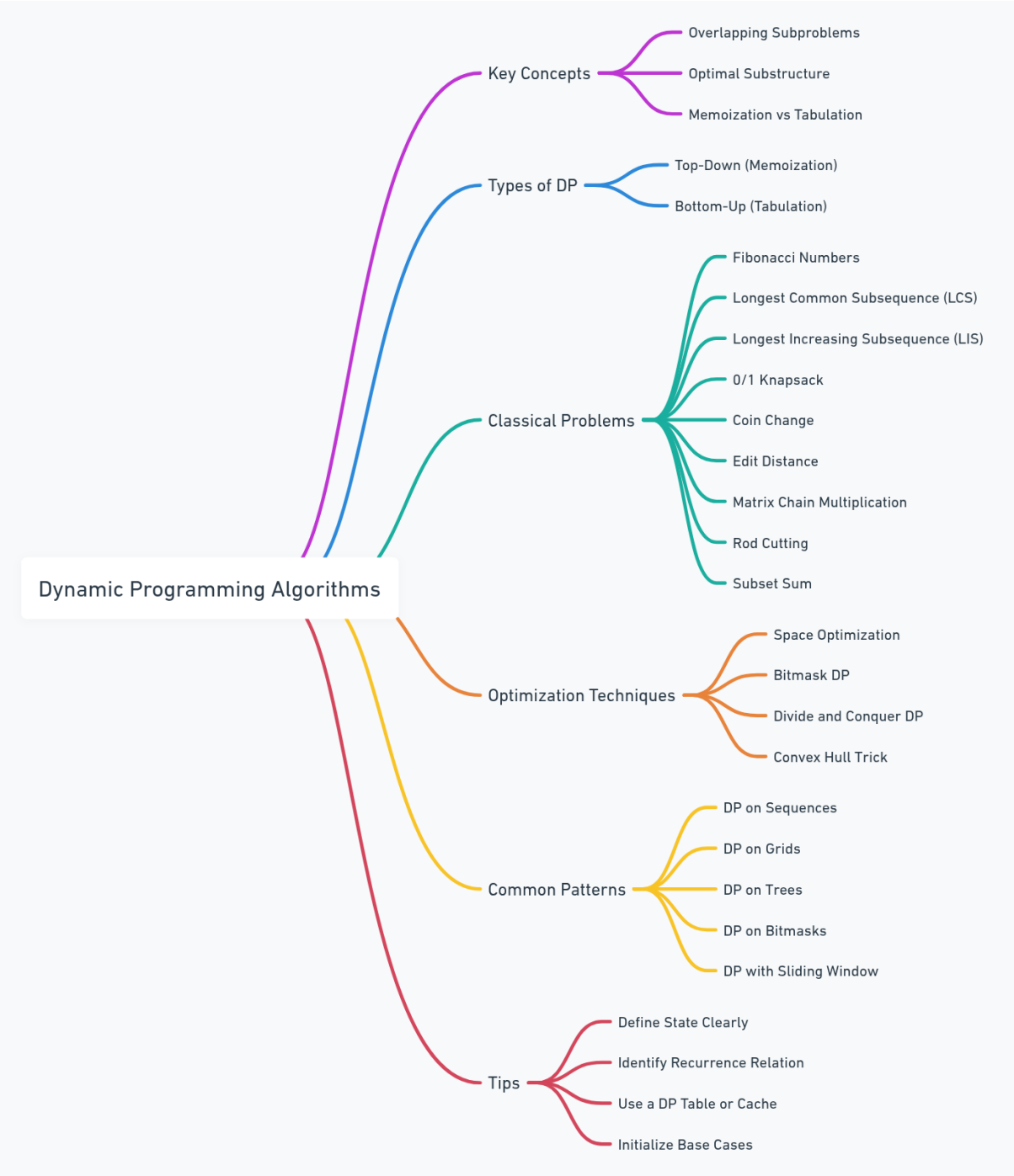


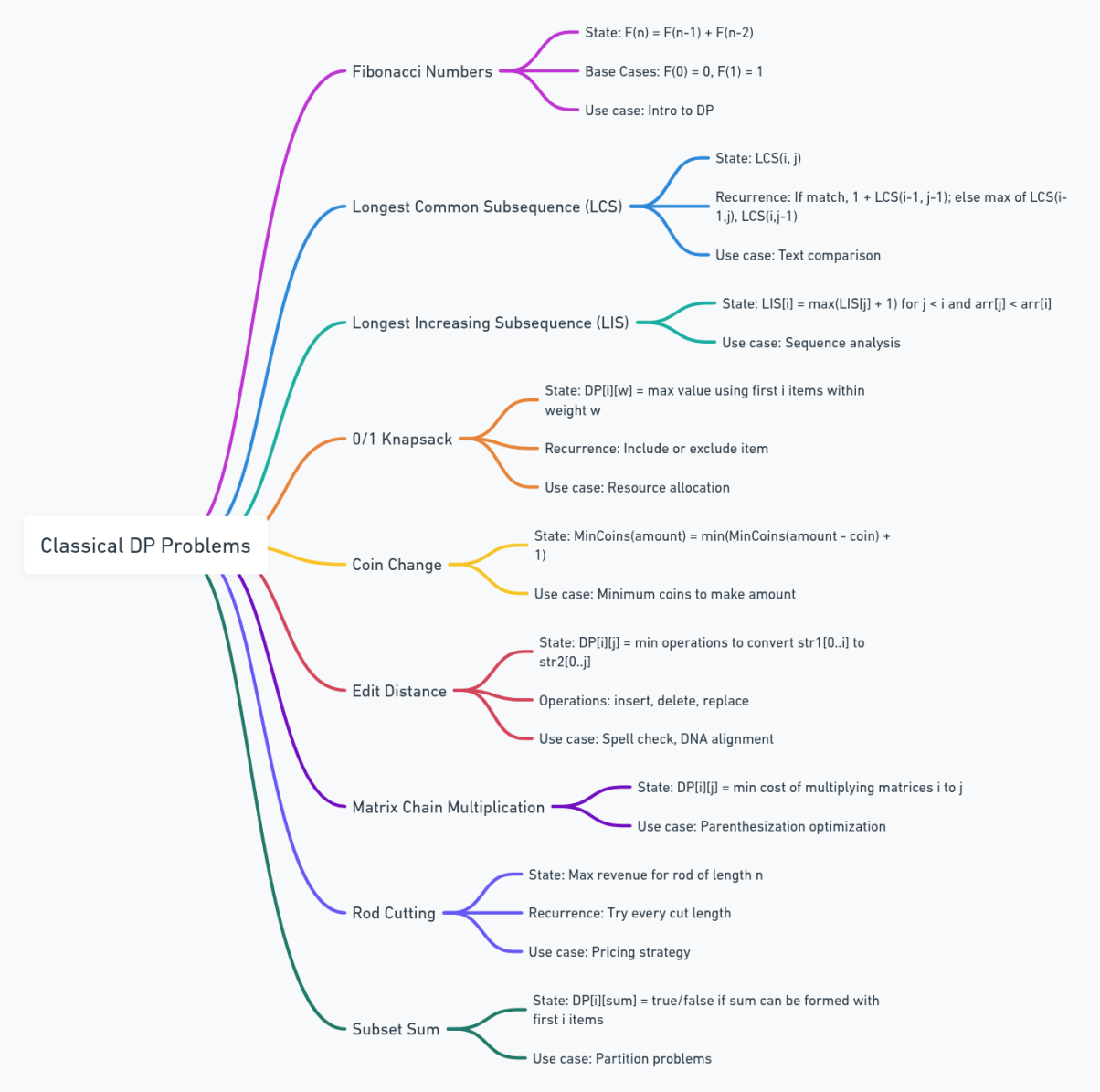
Greedy Algorithm



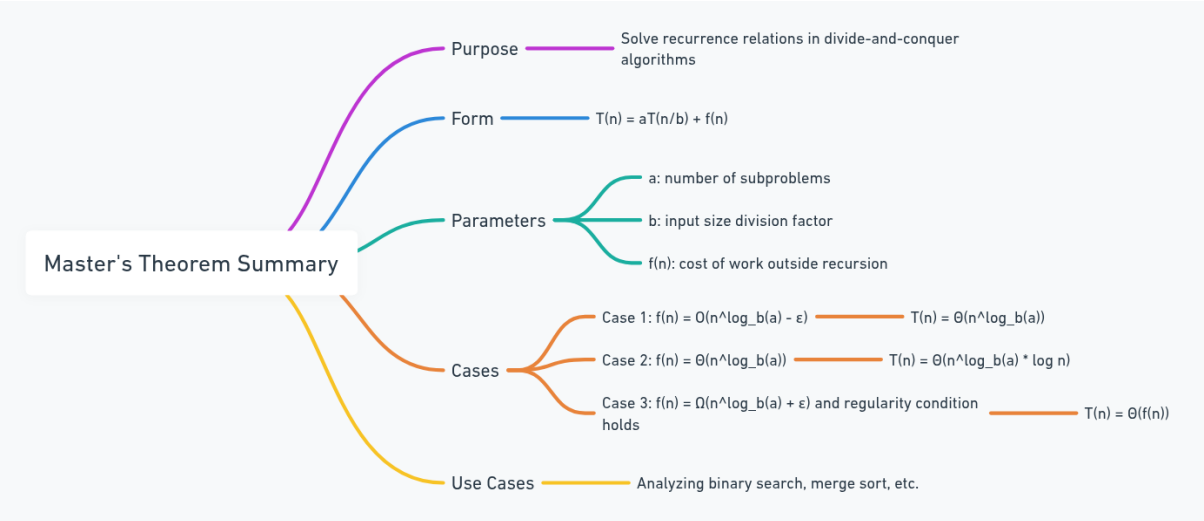


Dynamic Programming





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

🔍 Rabin-Karp[8] Algorithm: Overview

The **Rabin-Karp algorithm**[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.

[8] https://chatgpt.com/c/68513368-4bb0-800c-8173-e6a73909a047
[9] "Introduction to Algorithm – CLRS, 4TH ed", Page 962

- 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] 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.

[10] https://chatgpt.com/c/6850e100-d448-800c-a1de-8eac43519715

📜 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	$O(n * log\, n)$	
Average	$O(n * \log n)$	
Worst	$O(n^2)$	

➢ **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

Step	Subarray	Pivot	Result After Partition
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] 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
```

[11] https://chatgpt.com/c/6850e962-4554-800c-9ec4-45559e5746a5

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] 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.

[12] https://chatgpt.com/c/6850ed1f-9e9c-800c-a6a1-3c77b1ccc736

📋 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/)—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)