Date: __/__/20 Day: _____

Written by: Mirza Yasir Abdullah Barg

DSA
Data Structure & Algorithms

Hand written Notes

Topics we will cover are:

- DSA Introduction
  - DSA History
  - DSA Simple Algorithms
- Arrays
  - DSA Arrays
  - DSA Bubble Sort
  - DSA selection Sort
  - DSA Insertion Sort
  - DSA Quick Sort
  - DSA Counting Sort
  - DSA Redix Sort
  - DSA Merge Sort
  - DSA Linear Search
  - DSA Binary Search
- Linked Lists
  - DSA Linked Lists
  - DSA Linked Lists in Memory
  - DSA Linked lists Types
  - DSA Linked list Operations.

Date: __/__/20__ Day: _____

- **Stack and Queues**
  - DSA Stacks
  - DSA Queues

- **Hash Tables**
  - DSA Hash Tables
  - DSA Hash Sets
  - DSA Hash Maps

- **Trees**
  - DSA Trees
  - DSA Binary Trees
  - DSA Pre-ordered Traversal
  - DSA In-ordered Traversal
  - DSA Post-ordered Traversal
  - DSA Array Implementation
  - DSA Binary Search Trees
  - DSA AVL Trees

- **Graphs**
  - DSA Graphs
  - Graphs Implementation
  - DSA Graphs Traversal
  - DSA Cycle Detection

- **Shortest Path**
  - DSA Shortest Path
  - DSA Dijkstra's Algorithm
  - DSA Bellman - Ford

Date: __/__/20 Day: _____

Minimum Spanning Tree

- DSA Minimum Spanning Tree
- DSA Prim's
- DSA KrusKal's

Maximum Flow

- DSA Maximum Flow
- DSA Ford- Fulkerson
- DSA Edmonds - Karp

Time Complexity

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Counting Sort
- Radix Sort
- Merge Sort
- Linear Search
- Binary Search

DSA Advanced.

- DSA Euclidean Algorithm
- DSA Huffman Coding
- DSA The travelling Saleman
- DSA 0/1 Knapsack
- DSA Memoization
- DSA Tabulation
- DSA Dynamic Programming
- DSA Greedy Algorithms

Date: __/__/20__ Day: _____

DSA - Introduction (Depth theriotical)

DSA (Data structures and algorithms) is the fundamental part of computer science that teaches you how to think & solve complex problem systematically.

- Using the right data structure & algorithms makes your program run faster, especially when working with lots of data.
- Knowing DSA can help you perform better in job interviews & land great jobs in companies.

Example:

```
my_array = [7, 12, 9, 4, 11]
minVal   = my_array[0]
for i in my_array:
    if i< minVal:
        mival = i
print ('Lowest value:', minVal)
```

What Should Already Know

DSA is not specific to any language. You should have basic understanding of any programming language.

- python
- C++
- C
- Java
- JavaScript.

Select any of programming language you are interested in and start learning DSA

Date: __/__/20__ Day: _____

DSA History

The word algorithm comes from 'al-Khwarizmi' named after a persion scolar who lived around 800.

The concept of algorithm problem-solving can be track back to ancient times, long after before the invention of computer

The study of DSA really took off with the invention of computers in 1940s, to efficiently mange & process data.

Today DSA is key part of computer Science equation & professional programming, helping us to create faster and more powerful software.

Introduction To Data Structures & Algo

Data Structures are ways to store & organize data
Algorithms are steps-by-step methods to solve problem using that data.
Together, DSA helps us handle large data and solve problems fasters.

What are Data Structures?

A data structure is a way to store and organize data based on what we have and whate we want to do with it.

Example:

A family tree help us see how people are related and easily find relatives, like ~~mom's~~ Your mother's mother. Without link in tree it would be harder to understand relationship.

Date: __/__/20__ Day: _____

Data Structures help manage large amounts of data efficiently, like in databases or search engine and are key to create fast algorithms.

Types of Data Structures

Primitive: Basic types like integer, float, chars & bool
Abstract: Built from primitive type e.g arrays, link lists, stacks, queues, trees, and graphs.

What are Algorithms?

An algorithm is a clear set of step-by-step instructions to solve a problem or reach a goal.
Example:

A cooking recipe is like an algorithm, it lists steps to make a dish. In computers, the steps are written with in programming language and instead of ingredients, we work with data.

- Algorithms are fundamental & foundation of programming. Good algorithms make programs faster and more efficient.

Example of algorithms in real life:

- GPS Finding the fastest route
- Cruise control in car or plane
- Search engine Finding results.
- Sorting movies by rating.

In programming, each algorithm is usually designed for a specific problem and works with certain data structures (e.g bubble sort works on arrays)

Date: __/__/20 Day: _____

Data Structures Together with Algorithms

Data Structure + Algorithms (DSA) work together

A data structure stores data, & algorithms let you search, or change or use that data efficienty

DSA is about finding the best way to store, access and process data to solve problems quickly.

By learning DSA, you can:

- Pick the right tool (data structure/algorithm) for Job
- Make program Faster & use less memory
- Solve complex problems step by step.

Where DSA is used?

DSA is everywhere in software:

- Handling huge data (social Networks, search engine)
- Scheduling tasks (deciding what run first)
- Planning routes (GPS shortest path)
- Optimizing processes (faster task completion)
- Solving complex problems (packing, machine learning)

Common Areas using DSA:

Operating Systems, Databases, web apps, machine learning, video games, cyptography, data analysis, search engines & many more.

Key Terms In DSA

- Algorithms:

  Step by step instructions to solve a problem

- Data_Structure:

  A way to store and organize data (eg arrays, link lists, trees).

Date: __/__/20 Day: _____

- Time Complexity:

  How Fast an algorithm runs as data size grows.

- Space Complexity:

  How much memory an algorithm uses.

- Big O Notation:

  A way to describe time/memory growth of an algorithm.

- Recursion:

  A function calling itself to solve smaller parts of a problem.

- Divide and Conquer:

  Breaking a big problem into smaller ones, solving each, & combining results.

- Brute Force:

  Trying all possible solutions to find best one.

Date: __/__/20 Day: _____

DSA Simple Algorithm

Fibonacci Number

The fibonacci sequence is a list of numbers wete:

- The first two numbers are 0 and 1
- Every next number = sum of 2 previous numbers.

example:

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

Why its important in programming

- Simple way to learn algorithms (step-by-step)
- Can be done with loops or recursion

Using Loop

We store last two numbers, add them, print the result, and update the number

```
prev2, prev1 = 0, 1
print(prev2)
print(prev1)
for i in range(18):
    newFibo = prev1 + prev2
    print(newFibo)
    prev2, prev1 = prev1, newFibo
```

Using Recursion

A function calls itself until we have the required numbers.

```
print(0)
print(1)
count = 2
def fibonacci(prev1, prev2):
    global count
```

Date: __/__/20__ Day: _____

```
if count <= 19:
    newFibo = prev1 + prev2
    print(newFibo)
    count += 1
    fibonacci(newFibo, prev1)
fibonacci(1, 0)
```

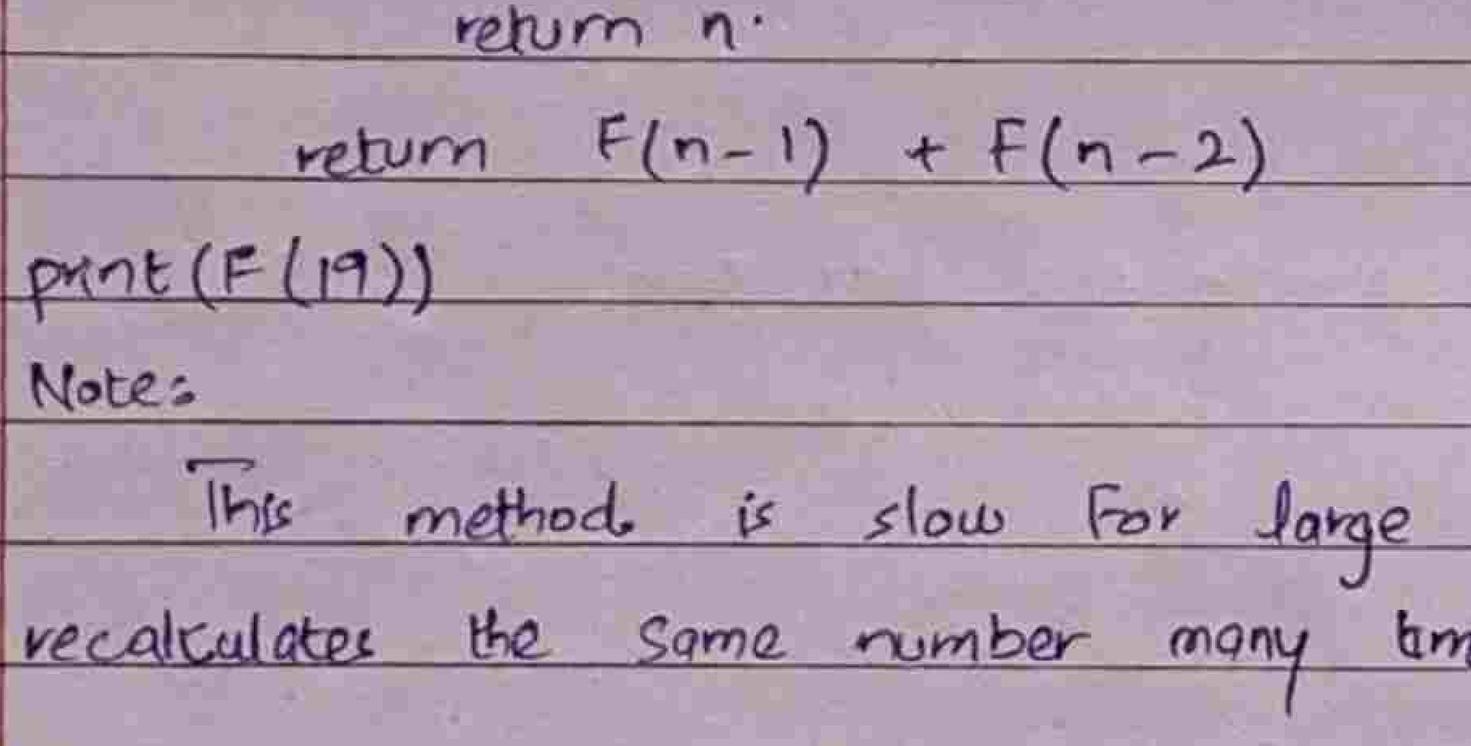
Finding the n-th Fibonacci Number (Recursive)

Formula: $F(n) = F(n-1) + F(n-2)$

```
def F(n):
    if n <= 1:
        return n
    return F(n-1) + F(n-2)
print(F(19))
```

Note:

This method is slow for large n, because it recalculates the same number many times.



Date: __/__/20 Day: _____

DSA Arrays:

What is an array?

- An array stores multiple elements in order
- Many algorithms use arrays (e.g finding the lowest value).

Python Example

my-array = [7, 12, 9, 4, 11]

Note:

(Python list vs array): In python, the code above creates a list (built-in dynamic array). For this, you can treat a list like an array.

Indexing (zero-based)

- Arrays are indexed. First element is at index 0 (python, Java, C all use zero-based indexing).

print(my_array[0])

Algorithm: Find the lowest value in an array

Idea (how it works):

- Look at the elements one by one.
- Keep track of the lowest so far.
- After checking all elements, the store lowest value it

Step-by-step (plain words):

- Create a variable minVal & set it to first element.
- Go through every element in array.
- If element is less than minVal, set minVal to element
- After the loop finishes, minVal is the lowest value.

Pseudocode:

```
minVal = array[0]
for each element in array:
    if element < minVal
        minVal = element
```

Date: __/__/20 Day: _____

Python Code

```
my_array = [7, 12, 9, 4, 11]
minVal = my_array[0]
for x in my_array:
    if x < minVal:
        minVal = x
print("Lowest value:", minVal)
```

What is pseudocode?

Pseudo code is half-human, half-code description that explains the logic without strict programming syntax. It helps you think clearly before writing real code.

Time Complexity (run time)

- The loop checks each element exactly once → O(n) (linear time)
  - 5 elements → 5 checks
  - 1000 elements → 1000 checks

About the "simulation/animation" you saw

- Some tutorials show an animation with:
  - Speed slider
  - "Find Lowest" button
  - "Lowest value" display
  - "Operations" counter (number of comparisons)
  - set values/Random/Ascending/Decending/1o random/Run/clear
- This controls do not change the algorithm. They only visulize how it scans the array and how many comparisons happen. This helps you see why the time is linear (O(n)).

Date: __/__/20__ Day: _____

Quick checklist (Remember this)

- Array = ordered collection, accessed by index.
- Zero-based indexing (first item is at 0).
- To Find the lowest, value, scan all elements & keep the minimum so far.
- Pseudocode helps you plan the algorithm.
- Python lists work like arrays For learning / Practice
- Time Complexity for this algorithm = O(n)

DSA Bubble Sort

What is Bubble sort?

- A sorting algorithm that arranges value from lowest to highest.
- The name comes from how large value "bubble up" to the end of the array.

How it works (concept)

- Go through the array one value at time.
- Compare each value with the next value.
- If the first value is bigger, swap them.
- Repeat this process many times until array is sorted.

Example (Manual Run → First Pass Only)

Start [7, 12, 9, 11, 3]

- Compare 7 & 12 → correct order, no swap.
- Compare 12 & 9 → swap → [7, 9, 12, 11, 3]
- Compare 12 & 11 → swap → [7, 9, 11, 12, 3]
- Compare 12 & 3 → swap → [7, 9, 11, 3, 12]

Observation:

The largest value 12 moved to the end.

Date: __/__/20 Day: _____

Pseudocode

```
for i from 0 to n-2:
    for j from 0 to n-i-2:
        if array[j] > array[j+1]:
            swap array[j] and array[j+1]
```

Python Implementation

```
my_array = [64, 34, 25, 12, 22, 11, 90, 5]
n = len(my_array)
for i in range(n-1):
    for j in range(n-i-1):
        if my_array[j] > my_array[j+1]
            my_array[j], my_array[j+1] = my_array[j+1],
                                         my_array[j]
print("Sorted array:", my_array]
```

Improved Bubble Sort (Early Exit)

If no swap happen in a pass → array is really sorted → stop early.

```
my_array = [7, 3, 9, 12, 11]
n = len(my_array)
for i in range(n-1):
    swapped = False
    for j in range(n-i-1):
        if my_array[j] > my_array[j+1]:
            my_array[j], my_array[j+1] = my_array[j+1], my_array[j]
            swapped = True
    if not swapped:
        break
print("Sorted array:", my_array)
```

Date: __/__/20 Day: _____

Time Complexity

- Worst case: $O(n^2)$ → Slow for large lists
- Best Case (already sorted with improvement): $O(n)$ → only 1 pas
- Why $O(n^2)$? outer loop runs n times, inner loop runs ~ n times → $n \times n$.

Key Notes

- Bubble Sort in simple but slow for large data.
- Useful for learning sorted concepts, but rarely used in production.
- Many faster algorithms exist (e.g QuickSort, Mergesort)

DSA Selection Sort

What is Selection Sort?

Selection sort is a way to sort a list by repeatedly finding the smallest number and moving it to the front.

How it works (step-by-step)

- look through the list to find the smallest number.
- Move that smallest number to the first position.
- Then, Look at the rest, of the list (ignoring the first one) and find the next smallest number
- Move that number to the second position
- Keep doing this until the whole list is sorted.

Example

[7, 12, 9, 11, 3]

Step 1: Find the smallest → 3 mae it to the front

[3, 12, 9, 11, 3]

Date: __/__/20 Day: _____

Step 2: Look at the rest → [7, 12, 9, 11] smallest is 7

→ no move needed.

[3, 7, 12, 9, 11]

Step 3: Next part → [12, 9, 11], smallest is 9:

[3, 7, 9, 12, 11]

Step 4: Last part → [12, 11], smallest is 11.

[3, 7, 9, 11, 12]

Done, Sorted.

Key Idea

We divide the list into two parts.

- Sorted part (at the start)
- Unsorted part (the rest)

Simple Python Implementation.

```
my_array = [64, 34, 24, 12, 22, 11, 90, 5]
n = len(my_array)
for i in range(n):
    min_idex = i
    for j in range(i+1, n):
        if my_array[j] < my_array[min_index]:
            min_index = j
    my_array[i], my_array[min_Index] = my_array[min_index],
                                       my_array[i]
print("Sorted array:", my_array)
```

Why Swap Instedd of Remove & insert?

If we remove & insert numbers.

- Other elements in the list must shift positions.
- This take more time.

Date: __/__/20 Day: _____

Better way: Swap the smallest number with the first unsorted number.
This is faster & avoids unnecessary shifting.

Time Complexity:
- Selection Sort checks almost every pair of elements.
- This means it takes about n^2 steps for n items
- Best Case: Already sorted → still $O(n^2)$ comparisons, but fewer swaps.
- Worst Case: Completely reversed → $O(n^2)$ comparisons & swaps.

Quick Summary
- Easy to understand.
- Not the fastest for large list.
- Always $O(n^2)$ time complexity.
- Works well for small lists.

DSA Insertion Sort

What is Insertion Sort?

Insertion sort works like sorting cards in your hand.
- You keep part of the list sorted.
- You take the next number from the unsorted part & insert it in the correct place in the sorted part.

How It works (Step-by-Step)

- Start: The first number is already "sorted" (because it's alone)
- Pick the next number from the unsorted part.
- Compare it to the numbers in sorted part, from right to left.
- Shift larger numbers to the right to make space.
- Insert the number in its correct position
- Repeat untill all numbers are sorted.

Date: / /20 Day:

Example:

[7, 12, 9, 11, 3]

Step 1: First element 7 → already sorted

Step 2: Pick 12 → 12 is bigger than 7 → stays there.

[7, 12, 9, 11, 3]

Step 3: Pick 9 → compare with 12 → Shift 12 right → insert 9 between 7 and 12.

[7, 9, 12, 11, 3]

Step 4: Pick 11, Shift 12 right → insert 11 b/w 9 & 12.

[7, 9, 11, 12, 3]

Step 5: Pick 3 → shift 12, 11, 9, 7 → insert 3 at start

[3, 7, 9, 11, 12] Done

Key Points

- The sorted part grows one element at a time.
- The unsorted part shrinks as we go.
- Works well for small lists or nearly sorted lists.

Python Implementation (Simple Way)

```
my_array = [64, 34, 25, 12, 22, 11, 90, 5]
n = len(my_array)
for i in range(1, n):
    current_value = my_array[i]
    insert_index = i
    for j in range(i-1, -1, -1):
        if my_array[j] > current_value:
            my_array[j+1] = my_array[j]
            insert_index = j
        else:
            break
    my_array[insert_index] = current_value
print("Sorted array:", my_array)
```

Date: __/__/20 Day: _____

Why this Version is Better

- Instead of removing & inserting (which shifts every thing twice), we only shift the elements needed.
- This saves Time and memory operations.
- In high-level langages (python, JS), removing & inserting causes hidden memory shifts, which can slow things down.

Time Complexity

- Worse Case (reversed list): $O(n^2)$ comparison & shifts
- Base Case (already sorted): $O(n)$ → just one check per elements
- Average case: $O(n^2)$

Summary Table

Case	Time Complexity	Space Complexity	Notes
Best	$O(n)$	$O(1)$	No shifting, comparisons
Average	$O(n^2)$	$O(1)$	Many shifts
Worst	$O(n^2)$	$O(1)$	Reversed order

DSA Quicksort

What is Quicksort?

Quicksort is one of the fastest sorting algorithms for general use. It works by picking a pivot element and rearranging the array so:

- All values less than pivot go to the left.
- All values greater than pivot go to the right.
- Then it recursively sorts the left & right sides.

How it Works (Step-by-step):

- Choose a pivot (here, we'll use the last element)

Date: __/__/20 Day: _____

- Partition: Move all smaller elements to the left of the pivot, and Larger ones to the right.
- Place Pivot: between left and right sides
- Recursively repeat for left & right sub-arrays
- Stop when sub-array has 0 or 1 elements

Example:

[11, 9, 12, 7, 3]

Step 1: Pivot = 3

All numbers are greater → swap 3 with first element (11)

[3, 9, 12, 7, 11] - 3 is sort correctly.

Step 2: Right side = [9, 12, 7, 11] → Pivot = 11

Move 7 to left 12 to right. Swap pivot into place.

[3, 9, 7, 11, 12] 11 in place.

Step 3: Left side of 11 = [9, 7] → Pivot = 7

Swap 9 & 7

[3, 7, 9, 11, 12] All Sorted.

Key Points

- Uses divide and conquer strategy
- Best when pivots splits array evenly.
- Works in place (no extra array needed).

Python Implementation

```
def _partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
```

Date: __/__/20__ Day: _____

```
        array[i+1], array[high] = array[high], array[i+1]
        return i + 1

def quicksort(array, low=0, high=None):
    if high is None:
        high = len(array) - 1
    if low < high:
        pivot_index = partition(array, low, high)
        quicksort(array, low, pivot_index - 1)
        quicksort(array, pivot_index + 1, high)

my_array = [64, 34, 25, 12, 22, 11, 90, 3]
quicksort(my_array)
print("Sorted array:", my_array)
```

Time Complexity

Case	Time Complexity	Space Complexity	Notes
Best	O(n log n)	O(log n)	Pivot splits array
Average	O(n, log n)	O(log n)	common scenario
Worst	O(n²)	O(log n)	smallest/largest.

Summary

- Very Fast for large datasets.
- Efficiency depends heavily on pivot change/choice.
- Often faster in practice than Merge Sort, despite same average complexity.

Date: __/__/20__ Day: _____

DSA Counting Sort

What is Counting sort?

Counting sort is a sorting method that doesn't compare numbers like Bubble Sort or Quick Sort.

It only works well for:

- Whole numbers (integers, not decimals)
- Non-negative numbers (0, 1, 2..... no negatives)
- When the largest number isn't too big compared to how many numbers you have.

When is Counting Sort fast?

- If the range of numbers (K) is small compared to the number of items (n).

Example: Sorting 1,000 numbers that are all between 0 and 50. (Fast)

- If K is much bigger than n, it's slow and wastes memory.

Example: Sorting 10 numbers that range from 0 to 1,000,000. (Not worth it).

How Counting Sort Works

Step 1 - Make a counting array

We first figure out the largest number in list.

Example: If the largest number is 5, we make a count array with indexes from 0 to 5.

Each spot will keep track of how many times that number appears.

Date: __/__/20__ Day: _____

Step 2: Count the numbers

We look at each number in our list and add 1 to the matching spot in the count array.

Example:

If the list is [2, 3, 0, 2, 3, 2]

The count array starts as: [0, 0, 0, 0]

- See 2 → count[2] becomes 1 → [0, 0, 1, 0]
- See 3 → count[3] becomes 1 → [0, 0, 1, 1]
- See 0 → count[0] becomes 1 → [1, 0, 1, 1]

Keep going until the count array is [1, 0, 3, 2]

Step 3: Rebuilt the sorted List

Now we go through the count array from start to end.

- For each index, add that number to the sorted list as many times as the count says.

Examples:

- count[0] = 1 → add 0 once → [0]
- count[1] = 0 → skip
- count[2] = 3 → add 2, 2, 2 → [0, 2, 2, 2]
- count[3] = 2 → add 3, 3 → [0, 2, 2, 2, 3, 3]

Done. This is sorted.

Code Example in python

```
def countingSort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)
    for num in arr:
        count[num] += 1
    sorted_arr = []
```

Date: __/__/20 Day: ____ ____

```
        for i in range (len(count)):
                sorted_arr.extend ([i] * count[i])
        return sorted_arr

unsortedArr = [4, 2, 2, 6, 3, 3, 1, 6, 5, 2, 3]
sortedArr = countingSort(unsortedArr)
print ("Sorted array:", sortedArr)
```

Time Complexity

- Best Case: O(n) where range (K) is small compared to n
- Worst Case: O(n+K) (which can be slow if K is huge)
- Needs extra space for the counting array.

Key points to Remember

- Works only for non-negative integers
- Very fast if the range of numbers (K) is small.
- Uses a counting array to track occurances
- Doesn't compare numbers at all.

DSA Radix Sort

Radix sort is a way to sort numbers by looking at one digit at a time, starting from the smallest place value (rightmost digit) & moving to the largest place value (lef most digit)

What is a redix?

The radix is the number of possible digits in our number system

- In our normal decimal system, there are 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

So, here, the redix = 10

Date: __/__/20__ Day: _____

How Radix sort Works?

- Start by looking at the rightmost digit (least significant digit)
- Group numbers into 10 buckets (one bucket for each digit from 0-9) based on that digit.
- Put the numbers back into the list in order of the buckets.
- Move to the next digit to the left and repeat the process.
- Keep going until you have sorted by leftmost digit.
- The list will now be fully sorted.

Why Radix Sort needs Stability?

A sorting algorithm is stable if numbers with the same digit keep their original order from before.

Examples:

if you have 45 & 22 (both end in 5), & 45 come before 25 in the list, they should still be in the same order after sorting by the last digit.

This is important because radix sort sorts one digit at a time. If we mess up the order in one step, the later steps won't work correctly.

Manual Example

Let's sort this

[33, 45, 40, 25, 17, 24]

Step No: 1

Sort by the last digit

Date: __/__/20__ Day: _____

Buckets:

0: [40]

1: []

2: []

3: [33]

4: [24]

5: [45, 25]

6: [ ]

7: [17]

8: []

9: []

put them back

[40, 33, 24, 45, 25, 17]

Step No: 2

Sorted by the next digit (tens place)

Buckets:

0: []

1: [17]

2: [24, 25]

3: [33]

4: [40, 45]

5: []

6: []

7: []

8: []

9: []

put them back

[17, 24, 25, 33, 40, 45]

Date: __/__/20__ Day: _____

Radix Sort Implementation Idea.

We need:

1- A list of numbers

2- A list of 10 empty bucks (0-9)

3- A loop to sort by each digit (starting from smallest place value)

4- A way to get the digit in focus:

- (number // exp) % 10

where exp is 1, 10, 100 ... for unit, ten, hund--

5- Repeat until we've processed all digits in largest number.

Example Code (Simple Version)

```
myArray = [170, 45, 75, 90, 802, 24, 2, 66]
print("Original array:", myArray)
maxVal = max(myArray)
exp = 1
while maxVal // exp > 0:
    buckets = [[] for _ in range(10)]
    for num in myArray:
        digit = (num // exp) % 10
        buckets[digit].append(num)
    myArray = [num for bucket in buckets for num in bucket]
    exp *= 10
print("Sorted array:", myArray)
```

Time Complexity

n = number of items to sort

K = number of digits in largest number (O(n*K))

Best case : O(n)

Worst case : $O(n^2)$

Average case : O(n log n)

Date: __/__/20 Day: _____

DSA Merge Sort

Merge Sort is a divide-and-conquer algorithm. It sorts an array by splitting it into smaller parts, sorting those parts, and then merging them back together in the right order.

Steps of Merge Sort

1- Divide

- Split the array into two halves
- Keep splitting until each part has only one element (a single element is always sorted)

2- Conquer (Merge)

- Merge the small parts back together in order.
- Always compare the first elements of each half and put the smallest one into merged list.
- Keep going until all parts are merged into one sorted array.

Example

Suppose we have:

[12, 8, 9, 3, 11, 5, 4]

Step-by-step split:

[12, 8, 9, 11, 5, 4]

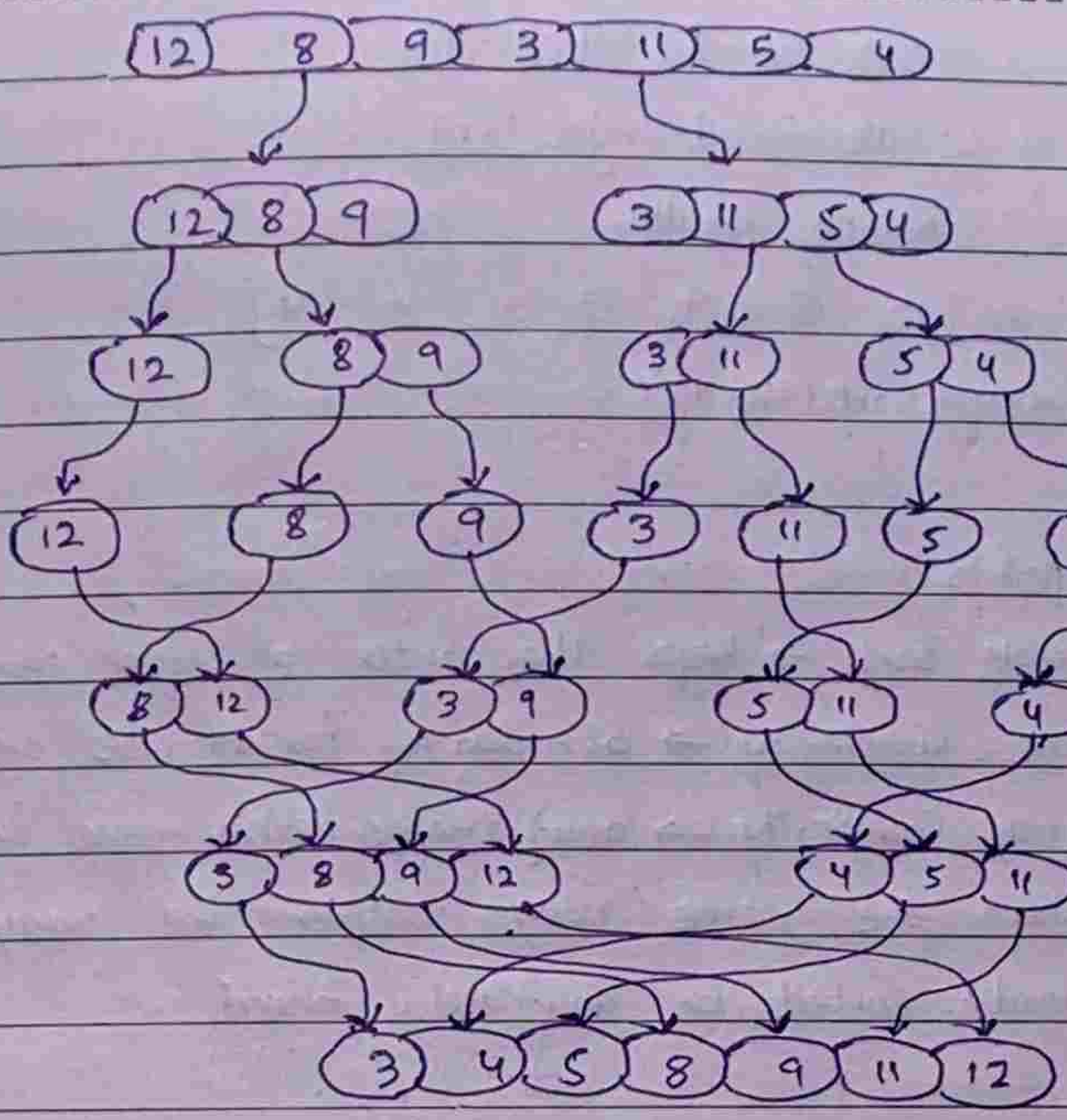
[12, 8, 9] [3, 11, 5, 4]

[12] [8, 9] [3, 11] [5, 4]

[12] [8] [9] [3] [11] [5] [4]

Step by step merge is discuss in the diagram

Date: __/__/20__ Day: _____



How it works in Code (Recursive Method)

```
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
    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
```

Date: __/__/20 Day: _____

```
        result.extend(left[i:])
        result.extend(right[j:])
        return result

data = [12, 8, 9, 3, 11, 5, 4]
print(merge_sort(data))
```

Key points

- Stable Sort → Keeps the order of equal elements
- Time Complexity → o(n log n) for all case (best, ave, worst)
- Space Complexity → o(n) (needs extra space for merge)
- Work the same way whether the array is already sorted or completely mixed.

DSA linear Search

What it is:

Linear Search is the simplest way to find something in a list (or array).
You check each ~~time~~ item one by one from the start until you find what you are looking for

When to use

- Works on any list (sort or unsorted)
- Easy to write and understand
- If the list is already sorted, Binary Search is faster

Important difference from sorting algorithms

- Sorting algorithms change the list order
- Searching algorithms (like linear Search) do not change the list → they only look through it.

Date: __/__/20__ Day: _____

How it works.

1- Start at first item in the list.

2- Compare it to value you wont to find. (target value)

3- if it matches → return the index (position) of item.

4- if it doesnt match → move to next item & repeat

5- If you reach the end of list & didnot find it, return 1 (mean "not found")

Example: Searching For 11

[12, 8, 9, 11, 5, 11]

Steps:

- Look at index 0 → value = 12 (not 11) → keep going
- Look at index 1 → value = 8 (not 11) → keep going
- Look at index 2 → value = 9 (not 11) → keep going
- Look at index 3 → value = 11 (found.)

Result: 11 is found at index 3

Stop Searching:

What if the Value is Not found?

If the loop finishes without finding it, we return -1.

Example:

Searching for 100 in [12, 8, 9, 11, 5, 11] returns -1

Python Code

```
def linearSearch(arr, targetVal):
    for i in range(len(arr)):
        if arr[i] == targetVal:
            return i

    return -1
```

Date: __/__/20__ Day: ____ ____

```
arr = [3, 7, 2, 9, 5]
targetVal = 9
result = linearSearch(arr, targetVal)
if result != -1:
    print(f"Value {targetVal} found at index {result}")
else:
    print(f"Value {targetVal} not found")
```

Time Complexity

- Best Case: The first item is target → only 1 comparison
- Worst case: target is last or not = n comparison
- Time Complexity = O(n)

It doesn't matter if the list is sorted, unsorted, random, ascending, or decending - linear Search will always check one by one.

DSA Binary Search

What it is:

Binary search is a method to quickly find a value in sorted array (list). It's inust faster than linear search because its keeps cutting the search area in half until the value is found (or not found).

Key points

- Works only on sorted data (smallest to largest or largest to smallest)
- Much foster than linear search for large lists.
- Uses the middile point to decide where to search next.

Date: __/__/20__ Day: _____

How its Work

- Look the middle value of array
- If the middle value is the target, return its index
- If the target is smaller, search only the left hand
- If the target is bigger, search only the right half
- Repeat the process until the value is found or search area is empty.

Example Search

We want to find 11 in this sorted list.

[2, 3, 7, 7, 11, 15, 25]

Step-by-step

1- Middle Index = (0+6) / 2 = 3 → value = 7

7 < 11 search right half [11, 15, 25]

2- Middle Index = (4+6) / 2 = 5 → value = 15

15 > 11 search half left [11]

3- Middle Index = (4+4) / 2 = 4 → value = 11

Found at index 4.

Why it's Fast

Each step halves the search area.

Example: In a list of 1,000 items, Binary Search only needs about 10 steps to find a value (because $\log_2(1000) \approx 10$).

Time Complexity

- Worst case: $O(\log_2 n)$ comparisons
- Much faster than linear Search's $O(n)$ for large lists.

Date: __/__/20__ Day: _____

Binary Search Code (Python)

```
def binarySearch(arr, target):
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

arr = [1, 3, 5, 7, 9, 11, 13]
target = 11
result = binarySearch(arr, target)
if result != -1:
    print(f"Value {target} found at index {result}")
else:
    print("Value not found")
```

Summary

- Binary Search is fast but needs sorted data
- Cuts the search area in half each time.
- Time Complexity: O(log n)
- Best For large datasets where speed matters.

Date: __/__/20__ Day: _____

DSA Linked Lists

A linked list is a sequence of connected nodes

Each node has:

- Data - the value it stores
- Pointer(Link) - the address of the next node in the list.

Example of a Singly Linked List

[Data | Next] → [Data | Next] → [Data | Next] → None

How it Works

- Nodes are sorted anywhere in memory
- The pointer in each node connects it to next node
- The last node points to None (end of list).

Benefits of Linked Lists.

- Flexible size → can grow or shrink easily
- Easy insertion & delection - just change pointers, no need to shift elements

Linked Lists Vs Arrays.

Features	Arrays	Linked Lists
Built in Structure	Yes	No
Fixed size in memory	Yes	No
Stored contigously in memory	Yes	No
Low memory usage	Yes	No
Dired access to element	Yes	No
Insert/Delete without shifting	No	Yes

When To Use linked lists

- When you dont know the final no. of elements.
- When you need fast insertions/delections.
  When memory is fragmented.

Date: __/__/20__ Day: _____

Drawback

- Uses more memory (because of pointers)
- Slower to access element (must start from the first node & follow links).

DSA Linked Lists in Memory

What is Computer Memory?

- When a program runs, it uses computer memory to store data like variable, arrays, & linked lists.
- Memory is divided into addresses (like numbered lockers). Each address stores data.
- Example: If an integer 17 is stored at memory address 0x7F25, the computer knows where to find it.

How Arrays are Sorted?

- Arrays store elements next to each other in memory (contigously).
- If each integer take 2 bytes, & the first element is at 0x7F23, then:
  - 1st element: 0x7F23
  - 2nd element: 0x7F25
  - 3rd element: 0x7F27 ... and so on
- Downside:

Adding or removing elements requires shifting all the elements after it. This takes time.

- Also: You must decide array size in advance (C)

How Linked Lists are sorted?

- A linked list is a collection of nodes.
  - Each nodes has:
    - Data (eg number 5)
    - Pointer to next node's address

Date: __/__/20__ Day: _____

- Nodes can be stored anywhere in memory - they don't have to be next to each other.
- The First node is called Head. The Last nodes pointer is null (no next node)

Advantages of Linked Lists

- Easy to insert or remove nodes - no shifting needed.
- Flexible size - can grow or shrink easily.

Disadvantage of linked lists

No direct access - to get the 5th Node, you must start at the head & follow pointers.

More Memory usage - each nodes stores an extra pointer

Linked list Example

If we store

3 → 5 → 13 → 2 → Null

Each Node has:

Data (like 3)

Pointer (address of next node)

Example in memory

Node 1: Data = 3, pointer = 0x1234

Node 2: Data = 5, pointer = 0x5678

Node 3: Data = 13, pointer = 0x9ABC

Node 4: Data = 2, pointer = null

Python Implementation

```
Class Node:
    def __init__(self, data):
```

Date: __/__/20 Day: _____

```
        self.data = data
        self.next = None
node1 = Node(3)
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)
node1.next = node2
node2.next = node3
node3.next = node4
current = node1
while current:
    print(current.data, end=" -> ")
    current = current.next
print("null")
```

Key Takeaways

- Arrays: Fast random access, fixed size, stored together in memory
- linked lists = Flexible size, stored anywhere, slower to access specific items
- Understanding linked lists help in learning stacks, queues, trees, & graphs.

Date: __/__/20__ Day: _____

DSA Linked Lists Types

linked lists are data structures made of nodes, where each nodes stores.

- Data (value)
- Pointers' to other node(s)

There are 3 main types of linked lists.

Singly Linked List

- Each node stores.
  - Data
  - Pointer to the next node
- Take less memory (only one pointer per node)
- Can only be traversed forward.

Example Diagram

[Data | Next] → [Data | Next] → [Data | Next] → Null

Python Example

```
Class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
node1 = Node(3)
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)
node1.next = node2
node2.next = node3
node3.next = node4
current = node1
while current:
        print(current.data, end = " → ")
        current = current.next
print("null")
```

Date: / /20 Day:

Doubly Linked List

- Each node stores:
  - Data
  - Pointer to the next node
  - Pointer to the previous node
- Uses more memory (two pointer per node)
- Can be transersed Forward and backward.

Example Diagram

null ← [prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] → Null

Python Example

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
node1 = Node(3)
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)
node1.next = node2
node2.prev = node1
node2.next = node3
node3.prev = node2
node3.next = node4
node4.prev = node3
print("Forward:")
current = node1
```

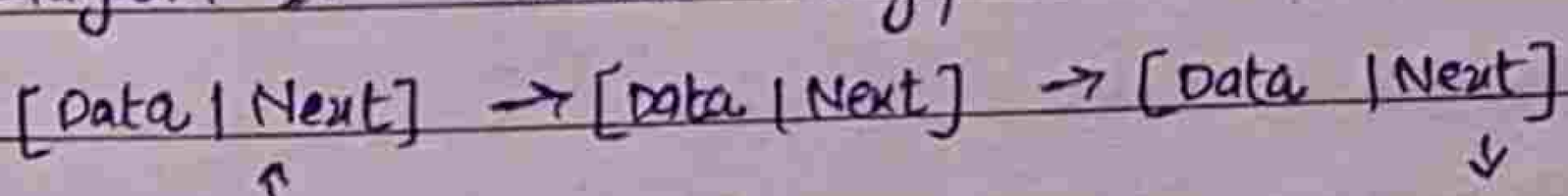
Date: __/__/20__ Day: _____

```
while current:
        print(current.data, end = " -> ")
        current = current.next
print("null")
print("Backward: ")
current = node4
while current:
    print(current.data, end = " -> ")
    current = current.prev
print("null")
```

Circular Linked List

- The last node points back to the first node instead of null.
- Can be:
  - Circular Singly Linked List: one pointer per node, Last node's next points to head.
  - Circular Doubly Linked list: two pointer per node, Last node next points to head, and head prev points to tail.
- No Null pointers, so special care is needed to avoid infinite loops

Diagram 1: Circular Singly Linked List



Python Example

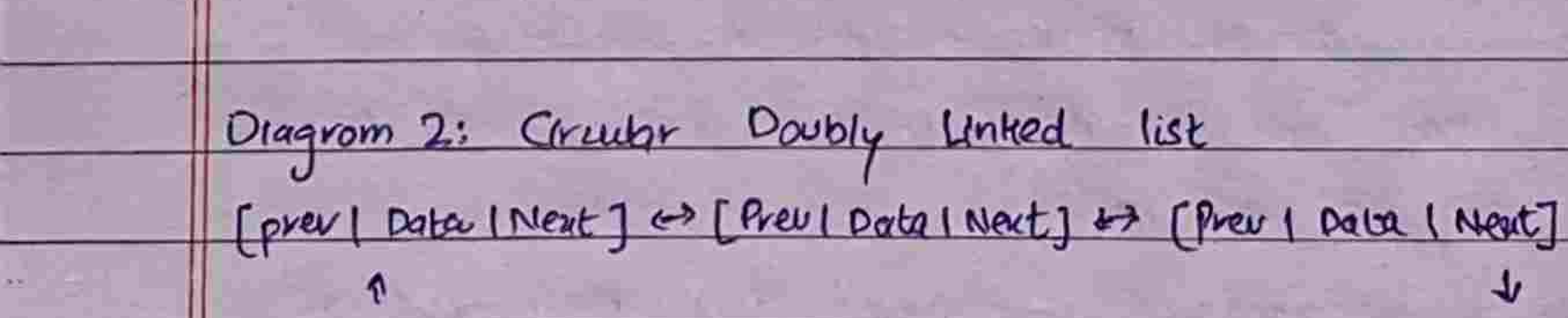
```
class Node:
    def __init__(self, data):
        self.data = data
```

Date: / /20 Day:

```
        self.next = None

node1   = Node (3)
node2   = Node (5)
node3   = Node (13)
node4   = Node (2)
node1.next  = node2
node2.next  = node3
node3.next  = node4
node4.next  = node1
current = node1
start = node1
print (current.data, end = " -> ")
current = current.next
while current != start:
    print (current.data, end = " -> ")
    current = current.next
print(" ... ")
```

Diagrom 2: Circular Doubly Linked list



Python Example

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
node1 = Node(3)
node2 = Node(5)
node3 = Node(13)
node4 = Node(2)
```

Date: __/__/20__ Day: _____

```
node1.next = node2
node1.prev = node4
node2.prev = node1
node2.next = node3
node3.prev = node2
node3.next = node4
node4.prev = node3
node4.next = node1
print("Forward:")
current = node1
start = node1
print(current.data, end=" → ")
current = current.next
while current != start:
        print(current.data, end=" → ")
        current = current.next
print("....")
print("Backend:")
current = node4
start = node4
print(current.data, end=" → ")
current = current.prev
while current.prev
        print(current.data, end=" → ")
        current = current.prev
print("...")
```

∴ A Table For all Thee Linked List

Date: __/__/20__ Day: _____

When to use which Linked List

Type	Memory Usage	Direction	BeSt for
Singly	Low	Forward only	Simple lists, stacks
Doubly	High	Forward & backward	Undo, redo
Circular	Low/High	continuous looping	Playlists

DSA Linked Lists Operations

A linked list is a data structure where each element (called a node) stores

1- Data

2- A link (pointer) to the next node in list

In singly Linked lists, each node only points to the next node.

Best Linked List Operations.

We can do many things with linked lists, but the most common are:

- Traversal - Go through all nodes in list
- Find the smallest value - Search for the lowest data value
- Delete a node - Remove a node from the list
- Insert a node - Add a new node anywhere in list
- Sorting - Arrange the nodes in a certain order

Traversal

What it means:

Visiting each nodes in the list, starting from the head, and following the links until we reach the end (null)

Uses:

- Search for a value
- Print the list
- Update a value

Date: __/__/20__ Day: _____

Example:

Head → 7 → 11 → 3 → 2 → 9 → Null

we follow each next link until null

Find the Lowest Value

We Look through every node, keeping tracking of the smallest value found so far.

Steps:

- Start with 1st node value as smallest
- Move to next nocle
- If the next node value is smaller, update smallest value.
- Countine until the end.

Deleting A node:

To delete a node:

- Find the node before the one you want to delete
- Change its next pointer to skip the node to delete
- Free the memory or let python garbage collect.

Important:

In a singly linked list, we must start at the head and move forward - we connot move backwards.

Insert a Node

To insert a new node

- Create the new node
- Find the position where it should be added.
- Change the next pointer of prev node to new node
- Point the new node to the next node in the sequence.

Date: __/__/20 Day: _____

Sorting

We can sort Linked Lists using algorithms like

- Selection sort
- Insertion sort
- Merge Sort

Note: Some Algorithms like counting sort or Quick Sort (index-based) don't work directly with linked list,

Time Complexity

- Traversal / Search: O(n) - same as array
- Insert / Delete at start: O(1)
- Insert / Delete at end: O(n) if singly link
- Sorting: Depends on algorithms (same big-O as array)
- Binary Search: Not possible without converting to array

Key Takeaways

- Linked lists are flexible in size
- They allow easy insertion & deletion
- They use more memory because of pointers.
- Traversal is required to find any element
- No random access like array

Date: __/__/20 Day: _____

DSA Stacks

What is stacks?

A stacks is a data structure that stores multiple elements but follows a special rule.

LIFO (Last in, First out)

- The last element added is the first one removed
- Example: A pile of pancake → you always take the top one first.

Basic Operations

- Push() → Add an element to the top
- pop() → Remove and return the top element
- peek() → Look at the top element (without removing)
- isEmpty() → check if the stack is empty
- size() → Get the number of elements

Stack Implementation Using Arrays (Python List)

- Pros: Easy, memory efficient, less code
- Cons: Fixed, size (can't grow beyond allocated memory)

```
stack = []
stack.append('A')
stack.append('B')
print(stack)

print(stack.pop())
print(stack[-1])
print(len(stack))
print(not bool(stack))
```

Date: __/__/20 Day: _____

Or Using a class:

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, x): self.stack.append(x)
    def pop(self): return self.stack.pop() if self.stack else "Empty"
    def peek(self): return self.stack[-1] if self.stack else "Empty"
    def isEmpty(self): return len(self.stack) == 0
    def size(self): return len(self.stack)
s = Stack()
s.push('A'); s.push('B')
print(s.pop())
print(s.peak())
```

Stack Implementation using Linked Lists

- Pros: Dynamic size (grows/shrink as needed)
- Cons: Uses extra memory (each nodes stores address)
  Longer code

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class Stack:
    def __init__(self):
        self.head( = None
        self.size = 0
    def push(self, value):
        node = Node(value)
        node.next = self.head
        self.head = node
        self.size += 1
```

Date: __/__/20__ Day: _____

```
def pop(self):
    if not self.head: return "Empty"
    val = self.head.value
    self.head = self.head.next
    self.size -= 1
    return val
def peek(self): return self.head.value if self.head
                    esle "Empty"
def isEmpty(self): return self.size == 0
def stackSize(self): return self.size
```

Uses of Stacks

- Undo/Redo in editors
- Backtracking (e.g maze soluing, recursion calls)
- Depth-First Search in graphs
- Expression evaluation (parentheses matching)

Summary: A stack is a simple but powerful data structure. You can built it using arrays (easy but fixed size) or linked lists (dynamic but memory heavy). Core idea = `Lifo (Last In, First out)

Date: __/__/20__ Day: _____

DSA Queues

A queue is a data store structure that stores element in order. It follows the rule:

FIFO (First In, First out)

- The first element added is the first one removed
- Example: People standing in line → person 1st come leave 1st

Basic Operations:

- enqueue() → Add element at the end(rear)
- dequeue() → Remove & return element from the front
- peek() → Look at front element (without removing)
- isEmpty() → check if queue is empty
- size() → Get number of elements

Queue Implementation Using lists (Array style)

- Pros: Easy to code, simple
- Cons: Removing from front (pop(0)) is slow (shifts element)

Code:

```
queue = []
queue.append(1)
queue.append(2)
queue.append(3)
print("Queue:", queue)
print("Dequeue:", queue.pop(0))
print("Peek:", queue[0])
print("is Empty:" len(queue) == 0)
print("Size:", len(queue))
```

Date: __/__/20

Day: _____

Queue Implementation Using collections.deque

Best way in python (fast at both ends)

Code:

```
from collections import deque
queue = deque()
queue.append(1)
queue.append(2)
print("Queue:", queue)
print("Deque:", queue.popleft())
print("Peek=", queue[0])
print("isEmply", len(queue)==0)
print("Size:", len(queue))
```

Queue Implementation Using Linked List

- Pros: Dynamic size, no shifting needed
- Cons: Extra memory (nodes), longer code

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class Queue:
    def __init__(self):
        self.front == self.rear = None
        self.size = 0
    def enqueue(self, value):
        node = node(value)
        if not self.rear:
            self.front = self.rear = node
        else:
            self.rear.next = node
```

Date: __/__/20 Day: _____

```
            self.rear = node
            self.size += 1
    def dequeue(self):
        if not self.front: return "Empty"
        val = self.front.value
        self.front = self.front.next
        if not self.front: self.rear = None
        self.size -= 1
        return val
    def peek(self): return self.front.value if self.front else "Empty"
    def isEmpty(self): return self.size == 0
    def queueSize(self): return self.size
```

Uses of Queues

- Printers (jobs handled in order)
- Task scheduling in OS
- Handling requests in web servers
- Breadth-First Search (BFS) in graphs
- Simulation of real-life waiting lines.

Summary: A queue is just like a waiting line FIFO (First In, First Out). Can be implemented with lists, deque or linked lists.

Date: __/__/20 Day: _____

DSA Hash Tables

A Hash Table is a special data structure that stores data in a way that searching, adding, and deleting very fast (O(1) on average)

- Think of it like a library index: instead of checking every book, you directly go to the right shelf using the index

Why Hash Tables?

- Array: Fast only if you know the index, otherwise you check each element.
- Linked List: Must search one by one → slow
- Hash Table: Uses a hash function to jump directly to the correct spot

Example: To find "Bob", the hash function calculates a number → tells us which bucket to check.

Key Concepts

- Buckets → Storage slots (like shelves)
- Key and value → data is stored using a unique key.
- Hash Function → converts the key into a number (hash code) → tells which bucket to use.
  - Example: "Bob" → Unicode sum → 275 % 10 = 5 → stored at index 5.

Main Operations

- add() → insert element into the right bucket.
- contains() → check if an element exists
- remove() → delete an element using its hash code
- Size() → count elements

Date: __/__/20 Day: _____

Collisions

Sometimes two keys map to the same bucket. Example "Lisa" and "Stuart" both go to the index 3.

To fix this:

- Chaining → Store multiple items in the same bucket
- Open Addressing → Find the next empty bucket.

Hash Set Vs Hash Map

Hash Set	Hash Map
Stores only unique keys	Stores key-value pairs
Example: Check if a name is on a guest list	Example: Store phone nos with names
Fast search, add, delete	Fast Search, add, delete.

Real life Uses

- Search Engines (stored and find words quickly)
- Databases (lookup tables)
- Compiliers (store variables)
- Caches (first retrieval)
- Checking membership (is "Alice" is the guest list?)

Summary

- Hash Table = data stored in buckets using hash function
- Very fast (o(1) average) for search, insert, delete
- Collisions are solved with chaining or open addressing
- Hash set = only keys, Hash Map = Key + Value.

In short: Hash Table is like a superfast dictionary for storing and finding data.

Date: __/__/20__ Day: _____

DSA Hash Sets

A Hash set is a special data structure that stores unique elements and lets you add, search, and remove very fast (average O(1) time)

- Think of it as magical box with labeled buckets. Each item goes into a bucket based on its hash code.

Key Ideas

- Unique Elements → No duplicates allowed
- Buckets → Containers to store elements
- Hash code → A no created from the value using a hash function (decides which bucket the item goes into)
- Collisions → If two items land in the same bucket, we just store them together inside that bucket (like a small list)

How it works

- Hash Function → Converts data (e.g "Peter") into number
  - Example: Sum of character codes → 512 % 10 = 2 → put "Peter" in bucket 2.
- Add → Store the item in the correct bucket.
- Contains → Recalculate hash code → check that bucket
- Remove → Find the item by hash code → delete it
- Size → Count elements

Time Complexity

- Best/Average Cases: O(1) (direct access)
- Worst case: O(n) if many items (and in same bucket)
- To avoid worst case: use a good hash Fonction + enough buckets.

Date: __/__/20 Day: _____

Real life Uses

- Checking membership (e.g is "Alice" in the guest list?)
- Removing duplicates From a collection
- Fast lookups in search engine, database, & caching

Code (Python)

```
class SimpleHashset:
    def __init__(self, size = 10):
        self.size = size
        self.buckets = [[] for _ in range(size)]
    def hash_function(self, value):
        return sum(ord(c) for c in value) % self.size
    def add(self, value):
        index = self.hash_function(value)
        if value not in self.buckets[index]:
            self.buckets[index].append(value)
    def contains(self, value):
        index = self.hash_function(value)
        return value in self.buckets[index]
    def remove(self, value):
        index = self.hash_function(value)
        if value in self.buckets[index]:
            self.buckets[index].remove(value)
    def print_set(self):
        for i, buckets in enumerate(self.buckets):
            print(f"Bucket {i}: {Bucket}")
hs = SimpleHashset(size = 10)
hs.add("Peter"); hs.add("lisa"); hs.add("Adele")
hs.print_set()
print("Contains Peter?", hs.contains("Peter"))
hs.remove("Peter")
Print("Contains Peter?, hs.contains("peter"))
```

Date: __/__/20 Day: _____

DSA Hash Maps

A Hash Map is like a dictionary

- Each entry has a key (unique id) and a value (the data linked to that key)
- Example: Social Security Number (key) → Person Name (value)

Important Terms

- Entry → a key-value pair (like 123-4567 : charlotte)
- Key → unique identifier (like social security no)
- Value → information linked to the key (name, address)
- Hash code → a no. created from key using hash fun
- Bucket → container where entires are stored, choosing using the hash code.

How it work

1. The hash function turns the key into a number.
   - Example for 123-4567 add digits → 28
   - Then do modulo 10 → 28 %- 10 = 8
   - This means the entry goes into bucket 8
2. When we search, add or remove:
   - The same hash code is calculated
   - That directs us to the right bucket directly

Operations (Fast)

- put(key, value) → add or update an entry
- get(key) → Find the value using the key
- remove(key) → delete an entry
- size() → number of entries in the map.

Date: __/__/20__ Day: _____

Time Complexity

- Normally: O(1) → super, fast, direct access
- Worst Case (all entries in same bucket): O(n)
- To stay fast, the hash function must spread entries evenly accross buckets

Why use Hash Maps?

- Faster than arrays/lists for lookups.
- Saves memory compared to using big array for large key
- Useful when each key must be unique(id, username)

Code (Python)

```
class SimpleHashMap:
    def __init__(self, size = 10):
        self.size = size
        self.buckets = [[] for _ in range(size)]
    def hash_function(self, key):
        numeric_sum = sum(int(ch) for ch in key if ch.isdigit())
        return numeric_sum % self.size
    def put(self, key, value):
        index = self.hash_function(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))
    def get(self, key):
        index = self.hash_function(key)
        for k, v in self.buckets[index]:
```

Date: __/__/20__ Day: _____

```
            if k == key:
                return v
        return None

    def remove(self, key):
        index = self.hash_function(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return

    def print_map(self):
        for i, bucket in enumerate(self.buckets):
            print(f"Bucket {i}: {bucket}")

hash_map = SimpleHashMap()
hash_map.put("123-4567", "Charlotter")
hash_map.put("123-4568", "Thomas")
hash_map.put("123-4570", "Peter")
print("Name for 123-4570:", hash_map.get("123-4570"))
hash_map.put("123-4570", "James")
print("Updated named for 123-4570:", hash_map.get
                                    ("123-4570"))
```

Date: __/__/20__ Day: _____

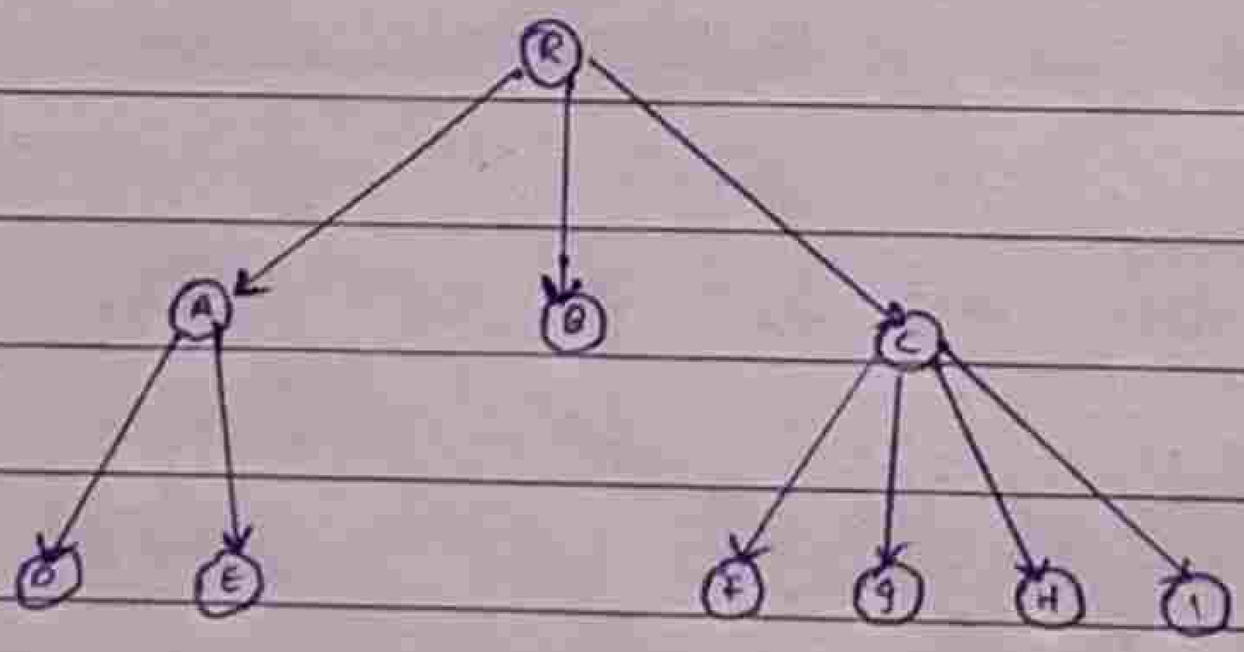
DSA Trees

A tree is a special data structure, like linked lists, but instead of being a straight line (linear), it branches like a real true tree upside down.

In linear structures (Arrays, Linked lists, Stacks, Queue), each element follows one after another. In trees, a single element can connect to many other elements.

Why Trees are Useful

- Hierarchical Data → File systems, company org. charts
- Databases → Fast searching and retrieval
- Networking → Routing tables
- Sorting / Searching → Algorithms Like binary Search Tree
- Priority Queues → Implemented with binary Heaps



Tree terminology

- Root node → The first/top node of tree
- Edge → A link b/w two nodes
- Parent Node → A node that has child nodes
- Child node → A node that comes From a parent
- Leaf node → A node with no children (end node)
- Tree height → No. of edges fromn root to farthest leaf
- Height of a Node → Edges From that node to farthest leaf
- Tree Size → Total no. of nodes in the tree

Date: __/__/20__ Day: _____

Example

If a tree has R and nodes A, B, C....

- R is the root
- R → A (edge)
- R is parent, A is child
- Nodes with no children = leave

Types of trees

- Binary trees: Each node has at most 2 children (left and right)
- Binary Search tree: (BST); A binary tree where:
  - Left children < Parent
  - Right children > Parent (Help in fast searching)
- AVL Tree: A self balancing BST
  - The difference in height b/w left & right subtree ≤ 1
  - If unbalanced, its fixes using rotations

•Summary:

A tree is hierarchical data structure with nodes connected by edges. It's powerful for fast searching, sorting, organizing, and sorting hierarchial data.

DSA Binary Trees

A binary tree is a speacal kind of tree where each node can have at most two children:

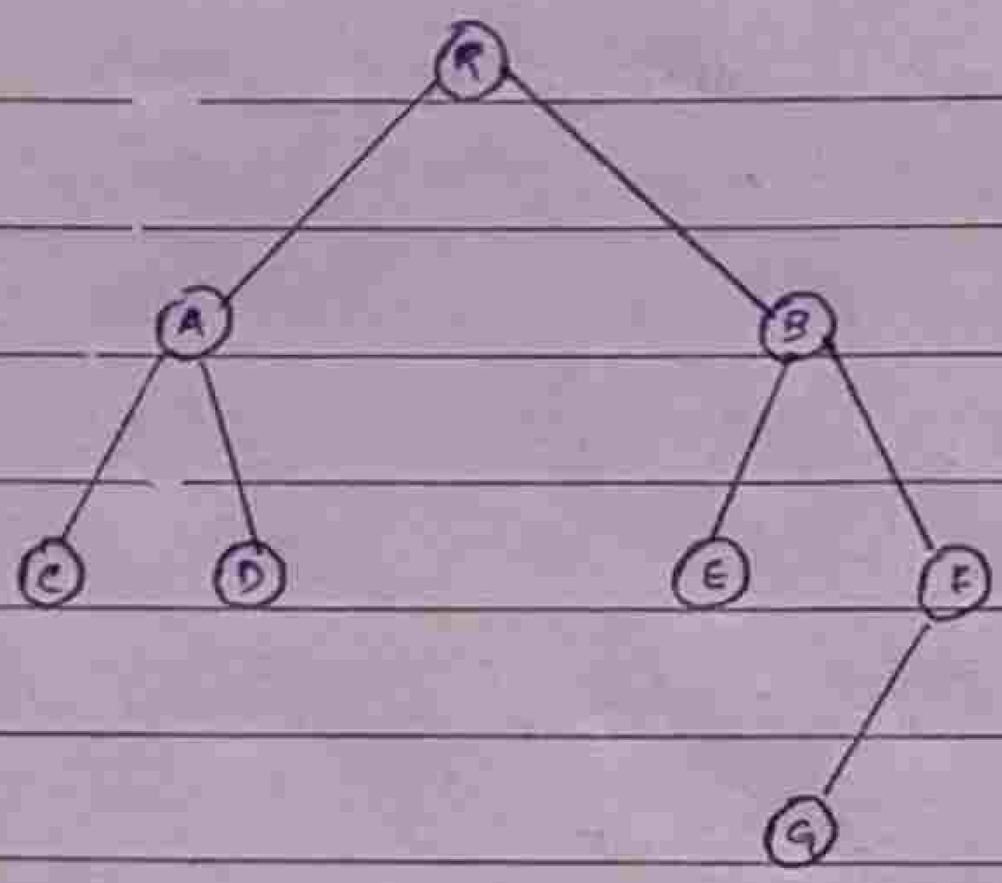
- Left child
- Right child

This restriction makes Binary trees eaiser to search, insert, delete and store efficientiy.

Why Tree are Useful?

Date: / /20 Day: ___ ___

- Easier algorithms for traversing, searching, inserting, deleting
- Binary Search Tree keeps data sorted → very fast searching
- AVL Tree (a balanced binary tree) keeps data well-organized
- Binary Trees can be stored in arrays → memory efficient



Binary Tree Vs Arrays and linked Lists

- Arrays → Fast direct access, but insertion/deletion is slow (shifting needed)
- Linked Lists → Fast insertion/deletion, but slow access (need to traverse)
- Binary Trees → Combine both: fast search + fast insert/delete (no shifting).

Types of Binary Trees

- Balanced Tree → Heights of left and right subtrees differ by at most 1.
- Complete tree → All levels are full except possibly the last, which is filled left to right.
- Full Tree → Every node has either 0 or 2 children
- Perfect Tree → All internal nodes have 2 children, and all leaf nodes are at the same level
  - A perfect tree is also full, balanced and complete.

Date: __/__/20__ Day: _____

Python

```
class TreeNode :
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
root = TreeNode ('R')
root.left = TreeNode ('A')
root.right = TreeNode ('B')
```

Binary Tree Traversal

Traversal = visiting all nodes

Two main categories

- Breadth First Search (BFS) → Visit level by level (sideway)
- Depth First Search (DFS) → Explore one branch fully before moving on.
    - Pre-order → visit root → Left → right
    - In-order → visit left → root → right
    - Post-order → visit left → right → root.

Summary:

Binary Trees are efficient, structured, and flexible combining the speed of arrays and the flexibility of linked lists. There are the foundation of BSTs, AVL, Trees, heaps, and many algorithms.

Pre-ordered Traversal (Binary Trees)

Pre-order traversal is a type of Depth First Search (DFS). In this traversal, nodes are visited in order:

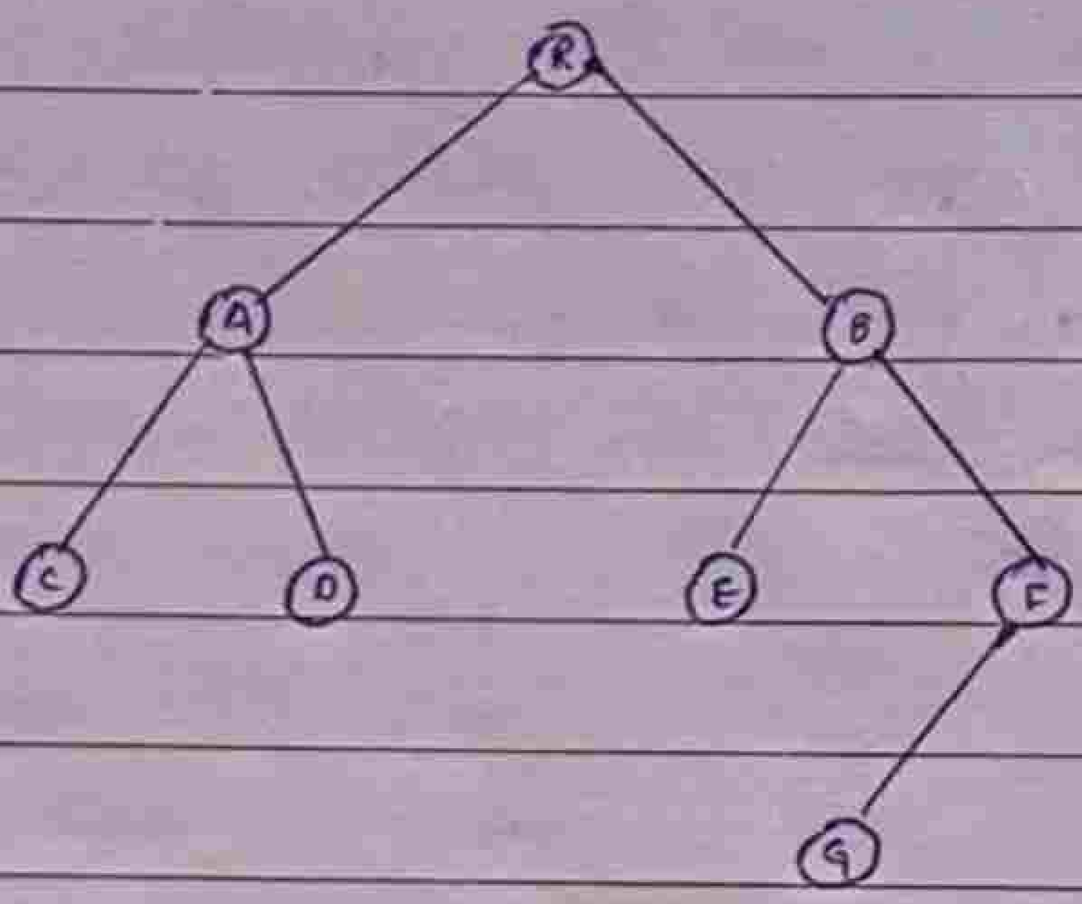
Root → Left subtree → Right subtree

Date: __/__/20__ Day: _____

How it works:

- Visit (or print) the current/root node first.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.

It is called 'pre'-order because the current node is visited before its children



Uses of pre-order

- Making a copy of tree
- Converting an expression tree into prefix notation

Python Example

```
def preOrderTraversal(node):
    if node is None:
        return
    print(node.data, end=", ")
    preOrderTraversal(node.left)
    preOrderTraversal(node.right)
```

How It Runs:

- Start at R → print it
- Go left to A, then C (left child)
- When none is reached (C has no left child), go back and move right
- Continue until the entire tree is traverse

Date: __/__/20__ Day: _____

In-order Traversal (Binary Tree)

In-order traversal is a type of Depth First Search (DFS)
In this traversal, nodes are visited in order.

Left Subtree → Root → Right Subtree

How it works:

- Recursively traverse the left subtree
- Visit (or print) the current/root node.
- Recursively traverse the right subtree.

It is called "in"-order because the node is visited in between left and right subtrees.

Uses of In-Order

- In binary search Tree, In-order traversal gives node in ascending sorted order.
- Useful for printing sorted data.

Python Example

```
def inOrderTraversal(node):
    if node is None
        return
    inOrderTraversal(node.left)
    print(node.data, end=", ")
    inOrderTraversal(node.right)
```

How It Runs

- Start at root R.
- Go left all the way until reaching C
- Print C (first node)
- Backtrack → print A, then go right → print D
- Return to root → Print R
- Finally visit right subtree → B, E

Date: __/__/20 Day: _____

Post-order Traversal (Binary Trees)

Post-order Traversal is a type of Depth First Search (DFS) used on binary trees. In this method, each node is visited after its left and right children.

Left Subtree → Right Subtree → Root Node

How it works

- Recursively traverse the left subtree
- Recursively traverse the right subtree
- Visit(print/use) the root node.

Uses of Post-order

- Deleting a tree
- Evaluating expressions in postfix notation (expression tree)

Python Example

```
def postOrderTraversal(node):
    if node is None:
        return
    postOrderTraversal(node.left)
    postOrderTraversal(node.right)
    print(node.data, end = ", ")
```

How It Run:

If a node C has no children:

- Left call → returns None
- Right call → returns None
- Then C is printed.

As recursion unwinds, each parent node is visited after its children, giving the final post-order sequence.

Date: __/__/20__ Day: _____

DSA Array Implementation

Normally, binary trees use pointers (left/right) to connect nodes. But trees can also be stored in an array, which can be more efficient in some cases.

Why use arrays?

- Pointers version is better when the tree changes a lot (insert/delete).
- Array version is better when tree is mostly read (less modify).
- Use less memory.
- Faster because of cache locality (array element are stored next to each other in memory, so CPU can access them faster)

How it Works

- Root node is stored at index 0.
- For a node at index i
  - Left child → index $2*i+1$
  - Right child → index $2*i+2$

Example: If node B is at index 2

- left child = $2*2+1 = 5 \rightarrow E$
- Right child = $2*2+2 = 6 \rightarrow F$

Important Note

Array implementation works best for perfect or nearly perfect trees (all levels filled except maybe the last) otherwise, it wastes space with None Values.

Traversal in Array Implementation

The same DFS traverals (pre-order, In-order, Post-order)

Date: __/__/20__ Day: ____ ____

work using recursive index calculations.

Example Functions

```
def pre-order(i):
    return [node] + left + right
def in-order(i):
    return left + [node] + right
def post-order(i):
    return left + right + [node]
```

Summary

- Arrays store tree nodes level by level
- Left child = 2*i+1, Right child = 2*i+2
- Efficient for perfect trees and read-heavy operations
- Traversals (pre/in/post) work same, just with index-based recursion

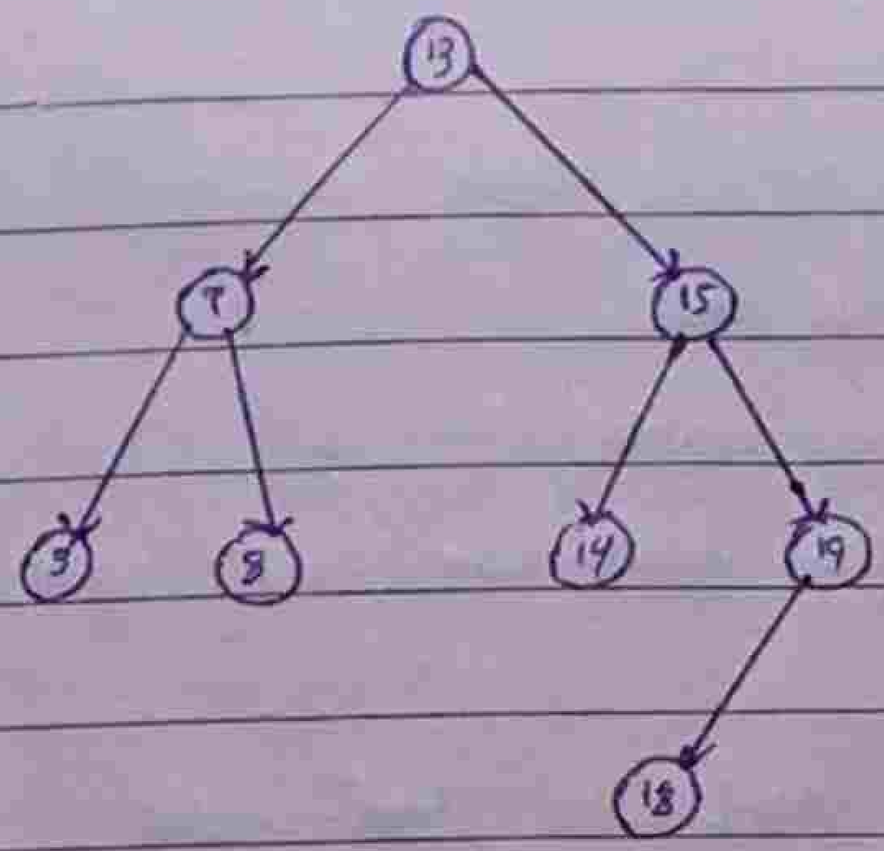
Binary Search Trees (BST)

A Binary Search Tree is a special type of Binary Tree with these rules:

- Every node left child (and its descendants) has a smaller value.
- Every node right child (and its decendants) has a larger value.
- Both left and right subtrees must also follow the same BST rules.
- All values are unique (assumption for simplicity)

This make searching, inserting, and deleting values faster than in a normal binary tree

Date: __/__/20__ Day: _____



- Size(n): Number of nodes (here 8)
- Height(h): Longest path from root to leaf (here 3)
- Root: Top node (13)
- Parent node: Node with children (like
- Leaf node: Node with no leaf (3, 8, 14, 18)
- Subtree: Any node and all its descendant (example: subtree at 15 includes 15, 14, 19, 18)
- In-order successor: Next node in in-order (13 → succ → 14)

Traversal of BST

- In-order traversal of BST alway gives value in sorted
- Example in-order of above tree: 3, 7, 8, 13, 14, 15, 18, 19
- This is one way to confirm if Binary Tree is BST

Searching in BST

works like Binary Search on sorted Array:

- Start at root
- If value matches → found
- If value is smaller → go left
- If value is larger → go right
- If subtree does'nt exist → value not found

Time Complexity: O(h) : where h = tree height

- Balanced tree: h = log(n) → fast } unbalnced tree = h = n → slow

Date: __/__/20 Day: ____ ____

Insertion in BST

- Similar to searching
  - If smaller → go left
  - If larger → go right
- keep going until you find an empty spot → insert new node there.
- New nodes always start as leaf nodes
- If value already exists → do nothing.

Finding the Minimum Value in Subtree

- Start at subtree root.
- Keep going left until no more child.
- That's the minimum value node.

Example:

- Minimum of subtree at 13 → 3
- Minimum of subtree at 15 → 14

Deleting in BST

Three Cases:

- leaf node - just remove it
- one child - replace node with its only child.
- Two children:
  - Find in-order successor (smallest node in right subtree)
  - Replace target node's value with successor's value.
  - Delete the successor node (it's guaranteed to be leaf).

Comparison with Arrays and linked List

Data Structure	Search	Insert/Delete
Sorted Array	O(log n)	Needs shifting
Linked List	O(n)	No shifting
BST (Balanced)	O(log n)	No shifting

Date: __/__/20__ Day: _____

BST combines the fast search of arrays and the easy insert/delete of linked lists

Balanced Vs Unbalanced BST

- Balanced BST → left and right subtrees differ in height by at most 1
  - Height ≈ log(n)
  - Operations: O(log n)
- Unbalanced BST → behaves like a linked list
  - Height ≈ n
  - Operations: O(n)

Summary:

- BST allows fast search, insert, and delete if it's balanced
- If unbalanced, performance worsens
- To keep height small, we use self-balancing trees (AVL trees, Red trees)

DSA AVL Trees

AVL Tree is a self-balancing Binary Search Tree (BST)
It is inverted in 1962 by Adelson-Velsky & Landis
It keeps the tree height small → so search, insert, and delete are always fast. Time complexity: O(log n) for search, insert, delete

Balance in AVL Tree

- A normal BST can get unbalanced (like a linked list), making operations slow (O(n))
- An AVL tree stays balanced by doing rotations whenever needed.

Date: __/__/20 Day: _____

- Balanced means:
  - Balance Factor(BF) = height (right subtree) - height(left subtree)
  - Values allowed: -1, 0, +1
  - If BF < -1 (too left heavy) or BF > +1 (too right-heavy), tree rotates.

Rotations:

Rotations are used to rebalance the tree while keeping BST rules intact.

- LL (Left-Left)
  - Node + left child are both left-heavy
  - Fix: Right Rotation
- RR (Right - Right)
  - Node + right child are both right-heavy
  - Fix: Left Rotation
- LR (Left - Right)
  - Node is left heavy, but its left child is right-heavy
  - Fix: Left Rotation on child → then Right rotation on node
- RL (Right - Left)
  - Node is right heavy, but its child is left-heavy
  - Fix: Right Rotation on child → then left Rotation on node.

Rebacing (After Insert / Delete)

- When you insert or delete, you move back up tree.
- At each ancestor, recalculate:
  - Height
  - Balance Factor
- If unbalanced → do the right rotation(s).
- Sometimes fixing one node also fixes higher nodes

Date: __/__/20__ Day: _____

AVL insert (Algorithm)

- Insert like normal BST
- Update nodes' height
- Calculate balance Factor
- If unbalanced - Perform the required rotation (LL, RR, LR, RL)

AVL Delete (Algorithm)

- Delete like normal BST (use minValueNode() for replacement)
- Update height and balance factor
- if unbalanced → perform the right rotation. (same rules as insertion).

Why AVL Trees are Fast?

- In a normal BST, worst-case = height ≈ n → oper (O(n)
- In an AVL Tree, height ≈ $\log_2(n)$
- So operations are O(log n)

Proof (simplified math):

- Perfect Binary Tree with height h has about 2^(h+1)-1 node
- So time Complexity : O(log n)

Summary

- AVL = BST + balance using rotations
- Keeps height low → garantees O(log n) search, insert, delete
- Uses rotations to fix 4 imbalance cases: LL, RR, LR, RL
- Balancing happens during retracing after insert/Delete

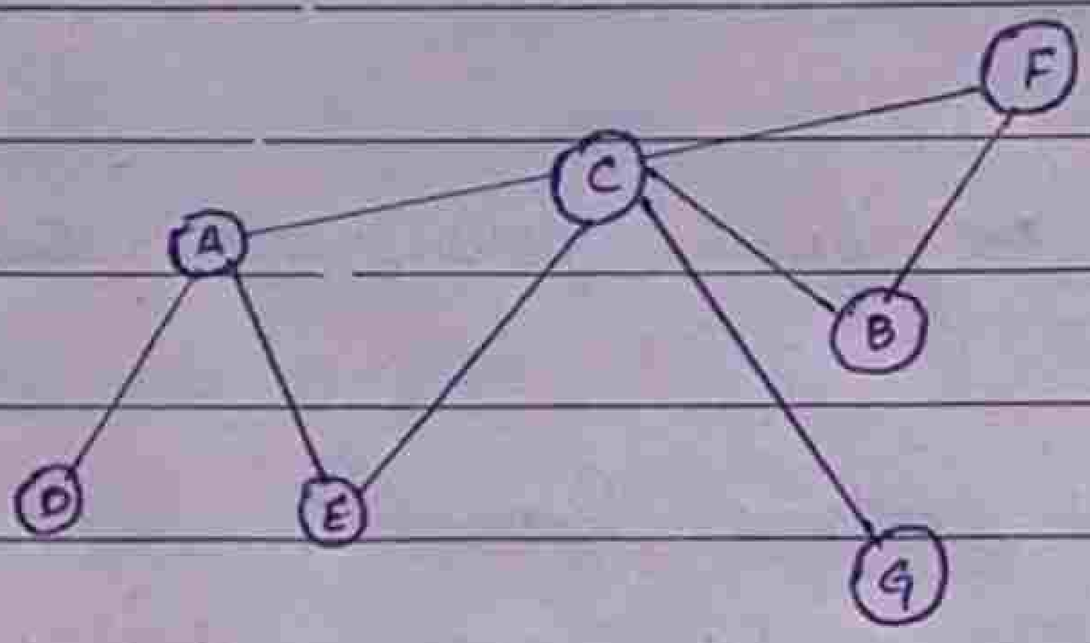
Date: __/__/20__ Day: _____

DSA Graphs

A graphs is a non-linear data structure made of:

- Vertices (node's): points / objects
- Edges: connections between vertices

Graphs are non-linear because there can be multiple paths between nodes (unlike arrays / linked lists)



Real-life uses of graphs

- Social networks: People = vertices, friendships = edges
- Maps and navigation: Cities / places = vertices, road, = edges,
- Internent: Webpages: vertices, links = edge
- Biology: Neurons, disease spread etc

Properties of graphs

- Weighted Graph: Edges have values (distance, cost, time)
- Connected graph: Every vertex is reachable from anothe if not it has isolated nodes or subgraphs.
- Directed graph (Digraph): Edges have a direction (A → B)
- Cyclic Graph:
  - Directed cyclic: You can follow arrows & return to start
  - Undirected cyclic: You can return to start without repeating edges
- Loop: An edge from a vertex to itself,

Graph Representations:

We need a way to store graphs in memory
Different methods affect space, speed and ease of use

Date: __/__/20 Day: _____

Adjacency Matrix

- A 2D Array (matrix) where:
  - matrix[i][j] = edge info from vertex i to vertex j
  - For unweighted graphs: use 1 (edge exist) or 0 (no edge)
  - For weighted graphs: store the weight instedd of 1
  - ~~For weighted graphs: matrix is symmetric~~
  - Undirected graphs: matrix is symmetric
  - Directed graphs: not necessarily symmetric

Good: Easy to understand and works for all graph type.

Bad: Uses a lot of memory (n^2 space)

Adjacency List

- An array of lists where each index = a vertex
- Each vertex stores a list to its neighbors (adjacent vertices)
- For weighted graphs, store (neighbor, weight)

Good: Saves space for sparse graphs (few edges)

Bad: slightly slower for checking if an edge exists.

Summary

- Graph = vertices + edge
- Types: weighted, Connected, Directed, Cydic, Loop
- Representations:
  - Adjacency Matrix: easy, but memory-heavy
  - Adjacency List: space efficient, best for sparse graphs

DSA Graphs Implementation

Before we use algorithms on a graph, we must first implement it. We usually use an Adjacency Matrix or ~~Not~~ Adjacency List. In this section, we'll focus on the Adjacency Matrix

Date: __/__/20 Day: _____

Adjacency Matrix Basics

- It's 2D array where each cell (i, j) shows if there is an edge from vertex i to vertex j
- For undirected graphs, edges go both ways, so the matrix is symmetric.
- Example: If A is vertex 0 and D is vertex 3, an edge b/w them is stored as 1 at (0,3) & (3, 0)

Example: (undirected, Unweighted Graph)

```
vertexData = ['A', 'B', 'C', 'D']
adjacency_matrix = [
        [0, 1, 1, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 1],
        [1, 0, 0, 0]
]
```

* we can also print connections more clearly

```
def print_connections(matrix, vertices):
    for _ in range(len(vertices)):
        print(f"{vertices[i]}:", end=" ")
        for j in range(len(vertices)):
            if matrix[i][j]:
                print(vertices[j], end=" ")
        print()
```

Using Classes (Better Implementation)

Instedd of raw arrays, we can use a Graph class

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
```

Date: __/__/20__ Day: _____

```
        self.vertex_data = [' '] * size

    def add_edge(self, u, v):
        self.adj_matrix[u][v] = 1
        self.adj_matrix[v][u] = 1

    def add_vertex_data(self, vertex, data):
        self.vertex_data[vertex] = data

    def print_graph(self):
        print("Adjacency Matrix:")
        for row in self.adj_matrix:
            print(row)
        print("\nVertex Data:", self.vertex_data)
```

Here, symmetry (line self.adj_matrix[v][u] = 1)

Directed and Weighted Graphs

For directed graphs, edges have directions, For weighted graphs, edges store weights instead of just 1

Changes

- Start with None instead of 0 for edges
- Add a weight parameter when adding edges
- Remove the symmebric line (self.adj_matrix[v][v] = weight)

So it's not automatically undirected.

Example (Directed, Weighted, Graph)

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[None] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [' '] * size
    def add_edge(self, u, v, weight):
        self.adj_matrix[u][v] = weight
```

Date: __/__/20__ Day: _____

```
def add_vertex_data(self, vertex, data):
    self.vertex_data[vertex] = data

def print_graph(self):
    print("Adjacency Matrix:")
    for row in self.adj_matrix:
        print([x if x is not None else 0 for x in row])
    print("Vertex Data:", self.vertex_data)
```

Example Usage

```
g = Graph(4)
g.add_vertex_data(0, 'A')
g.add_vertex_data(1, 'B')
g.add_vertex_data(2, 'C')
g.add_vertex_data(3, 'D')

g.add_edge(0, 1, 3)
g.add_edge(0, 2, 2)
g.add_edge(3, 0, 4)
g.add_edge(2, 1, 1)
g.print_graph()
```

Summary

- Adjacency Matric → 2D array representation
- Undirected Graph → matrix is symmetric
- Directed Graph → no symmetry, edges go one way
- Weighted Graph → edge values stores weights instead of 1
- Graph Class → better abstraction for vertices, edges, and methods.

Date: __/__/20 Day: _____

DSA Graphs Traversal

Traversal = visiting all vertices of a graph in a symmetric way. We usually want to start from one vertex and then move along edges untill all reachable vertices are visited.

Two Common Traversals

- DFS (Depth First Search) → goes deep first.
  - Uses stack (or recursion with call stack)
- BFS (Breath First Search) → goes level by level
  - Uses queue

Call Stack Reminder

When one function calls another

- New function goes on top of stack
- When its finishes, it's removed, and previous Func. continues

DFS uses this mechanism to go deep.

Depth First Search (DFS)

Start at a vertex, keep visiting an unvisited neighbor until stuck, then backtrack.

Steps:

- Pick a start vertex
- Mark it visited
- Recursively visit all unvisited neighbors.

Example Code

```
def dfs_util (self, v, visited):
    visited[v] = True
```

```
        print(self.vertex_data[v], end = ' ')
        for i in range(self.size):
            if self.adj_matrix[v][i] == 1 and not visited[i]:
                self.dfs_util(i, visited)
def
dfs     dfs(self, start_vertex_data):
            visited = [false] * self.size
            starte_vertex = self.vertex_data.index(start_vertex_data)
            self.dfs_util(start_vertex, visited)
```

Notes:

- visited array ensures we don't revisited nodes
- Recursion handles going 'deep'
- Runs until all connected vertices are visited.

Breath First Search (BFS)

Visit all neighbors of a vertex before moving furthe

Steps:

- Put start vertex in queue
- Take vertex from queue, mark visited, print it
- Add all unvisited neighbors to queue
- Repeat until queue is empty

Example Code

```
def bfs(self, start_vertex_data):
    queue = [self.vertex_data.index(start_vertex_data)]
    visited = [false] * self.size
    visited[queue[0]] = True
    while queue:
        current = queue.pop(0)
        print(self.vertex_data[current], end = ' ')
        for i in range(self.size):
            if self.adj_matrix[current][i] == 1 and not visited[i]
```

Date: __/__/20__ Day: _____

queue.append(i)

visited[i] = True

Notes:

- BFS processes vertices in layers (distance from start)
- Queue ensures first-in, first-out order

DFS and BFS on Directed Graphs

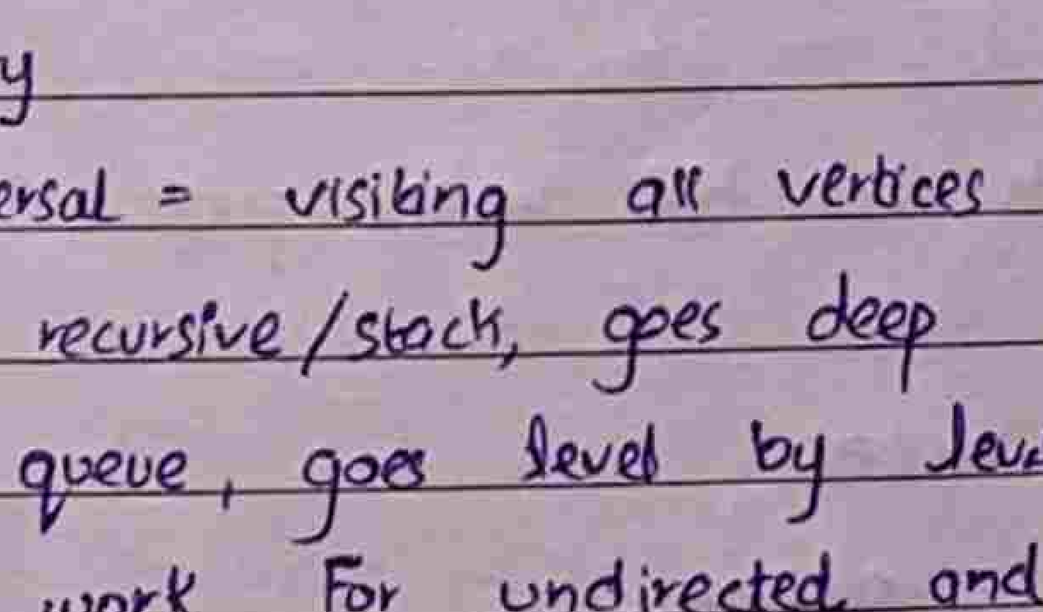
- Works the same way
- Just change add_edge() so it only sets self.adj_matrix[u][v] = 1 (no symmetric edge)
- Be careful when building the graph, since edge now have directions

Key Differences: DFS vs BFS

Aspect	DFS	BFS
Data Structure	Stack / Recursion	Queue
Strategy	Go deep first	Go level by level
Uses	path finding, topological sort	shortest path, level order

Summary

- Traversal = visiting all vertices
- DFS: recursive / stack, goes deep
- BFS: queue, goes level by level
- Both work for undirected and directed graphs with small adjustments



Date: __/__/20 Day: _____

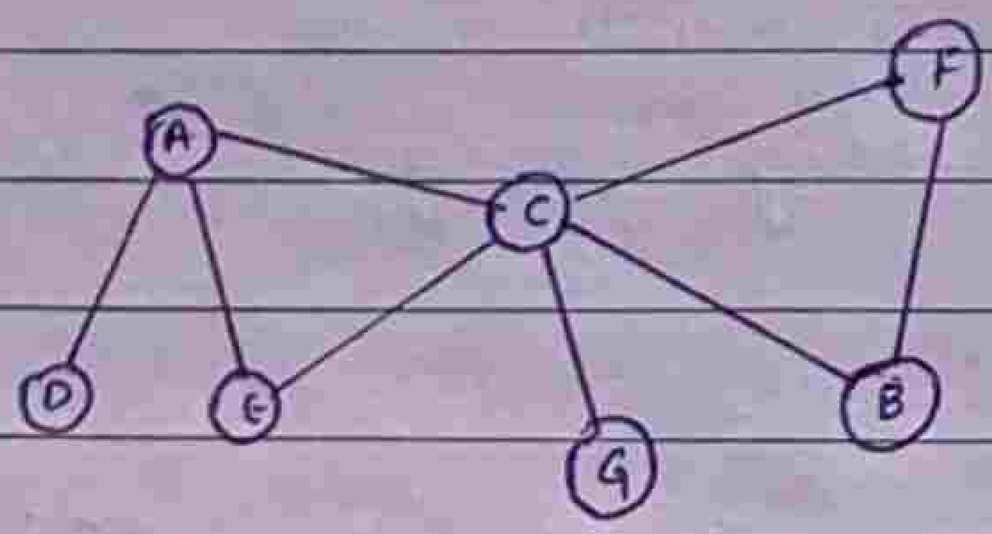
DSA Graphs Cycle Detection

Cycle = A path that starts and ends at the same vertex without repeating edges.

Example: Walking in a maze and ending up where you started.

Why Important?

- Cycles can cause problems in networking, scheduling and circuit diagram.
- We must detect them.



Methods to detect Cycles

- DFS (Depth First Search)
- Union - Find (Disjoint Set Union - DSU)

DFS Cycle Detection (undirected Graphs)

Steps:

- Start DFS from each unvisited vertex (graph may not be connected)
- Mark vertex as visited
- For each adjacent vertex:
  - If not visited, recursively DFS
  - If already visited and not parent, cycle found
- If DFS completes with no such case → no cycle

Key Idea: A visited neighbor that is not the parent means there is back-edge → cycle

Date: __/__/20__ Day: _____

Example (Python)

```
def dfs_util(self, v, visited, parent):
    visited[v] = True
    for i in range(self.size):
        if self.adj_matrix[v][i] == 1:
            if not visited[i]:
                if self.dfs_util(i, visited, v):
                    return True
            elif parent != i:
                return True
    return False
```

is_cyclic() → Runs DFS for all components and returns True if any cycle is found.

DFS Cycle Detection (Directed graph)

In directed graphs, a visited node doesn't always mean a cycle

Fix: Use an extra array recStack (recursion stack)

- recStack[v] = True → Vertex is part of current DFS path
- If we find an adjacent vertex already in recStack → cycle found
- When DFS, backtracks, mark recStack[v] = False

Example Python

```
def dfs_util(self, v, visited, recStack):
    visited[v] = True
    recStack[v] = True
    for i in range(self.size):
        if self.adj_matrix[v][i] == 1:
            if not visited[i]:
                if self.dfs_util(i, visited, recStack):
                    return True
```

Date: __/__/20 Day: _____

```
            elif resStack[i]:
                return True
        recStack[v] = False
        return False
```

Key Ideas:

- Undirected → check parent
- Directed → check recursion stack

Union Find Cycle Detection (Undirected Graphs Only)

Steps:

- Initially, each vertex is in its own subset
- For each edge (u, v):
  - Find roots of u and v:
  - if roots are the same → cycle detected
  - Else → Union the subsets.
- Stop as soon as a cycle is found.

Example Python

```
def find(self, i):
    if self.parent[i] == i:
        return i
    return self.find(self.parent[i])
def union(self, x, y):
    x_root = self.find(x)
    y_root = self.find(y)
    self.parent[x_root] = y_root
```

- find(x) → gets root of subset for x
- union(x, y) → merges two subsets.
- if both vertices already have the same root → cycle

Date: __/__/20 Day: _____

Quick Comparison

Method	Works For	Idea
DFS (Undirected)	Undirected graphs	Cycle = visited neighbor nor parent
DFS (Directed)	Directed graphs	Cycle = node revisited in same recursion path
Union-Find	Undirected graphs	Cycle = two vertices already in same set.

Summary

- A cycle = closed path
- Undirected DFS → check parent
- Directed DFS → use recursion stack.
- Union-Find → detect by checking if vertices already belong to same subset

DSA Shortest Path

The shortest path problem mean finding the path with the smallest total weight (or cost) between two vertices in a graph.

- Vertices = points (cities, routes, intersections)
- Edges = connections (road, flight paths, data links)
- Edge weight = distance, cost or time
- Path weight = sum of edge weights along path

Example: From D to F, shortest path = D → E → C → F with cost 10. other paths exist, but with higher total cost.

Solutions:

Two famous Theorem:

- Dijkstra Algorithm → works only when all edges weights are positive

Date: __/__/20 Day: _____

Bellman Ford Algorithm → works with negative edge weights and can detect negative cycles

Key Ideas:

- Start by setting all distances = ∞ (infinity)
- Distance from start node to itself = 0
- Keep checking edges:
  - If a shorter path is found → update distance
  - This is called relaxation (relaxing an edge)
- Repeat until all shortest paths are found

Positive Vs Negative Weights

- Positive edge: mean you "pay" something (distance or cost)
- Negative edge: mean you "gain" (earning money instead of lossing)
- Dijkstra cannot handle negatives but Bellman do.

Negative Cycles

- A negative cycle = loop where total weight < 0
- Example: A → E → B → C → A = -1 total cost
- Problem: you can keep lossing to reduce cost forever → shortest path doesn't exists.
- Bellman. Ford can detect negative cycles.

Summary

- Dijkstra = fast, positive weights only
- Bellman - Ford = slower, works with negatives, detects negative cycles.
- Negative cycle present? shortest path is impassible

Date: __/__/20 Day: _____

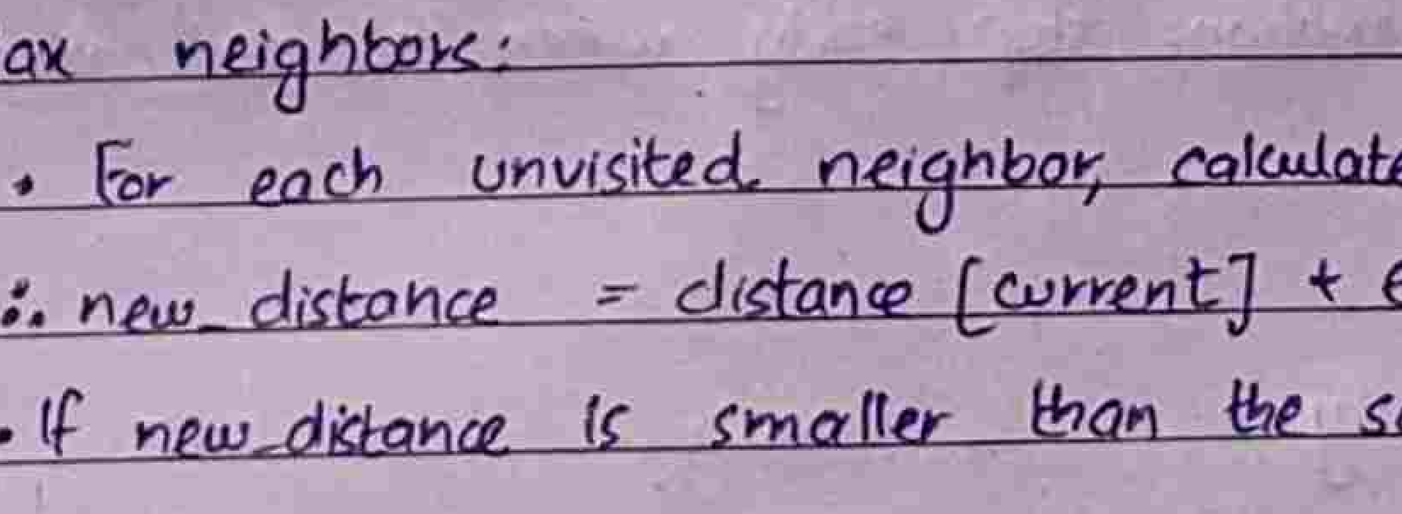
DSA Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from one starting vertex (source) to all other vertices in a graph

Work on different directed or undirected graphs, but not with negative edge weights (use Bellman-Ford for that)

How it works

- Initialize distances
  - Distance to source = 0
  - Distance to all others = ∞ (infinity)
- Pick the nearest unvisited vertex → make it the "current vertex"
- Relax neighbors:
  - For each unvisited neighbor, calculate
  ∴ new_distance = distance [current] + edge_weight
  - If new_distance is smaller than the sorted distance update it
- Mark current vertex as visited (if won't be checked)
- Repeat steps 2-4 until all vertices are visited



Example:

Say we start from D

, initially

distance = [A = ∞, B = ∞, C = ∞, D = 0, E = ∞, F = ∞, G = ∞]

Date: __/__/20__ Day: _____

- First, update neighbors of D → A=4, E=2
- Next, choose E(distance = 2). Relax → update C=6, G=7
- Next, choose A(distance = 4). Relax → no better update
- Next, choose C(distance=6). Relax → update F=11, B=8
- Next, choose G(distance=7). Relax → no update
- Next, choose B(distance= 8). Relax → update F=10
- Finally, F(distance =10). Done

Result: Shortest distances From D to all vertices

Variants

- Undirected graphs → add edges both ways
- Directed graphs → only one direction
- Returning Paths → keep a predecessor array to trace the shortest route (e.g, D → E → C → B → F)
- Single Destination: stop early when the destination is reached

Time Complexity

- Naive Version (array Search)

Time = $O(V^2)$

- Using Min-heap (priority queue)

Time = $O((V + E) \log V)$

Much better for large/sparse graphs

Summary

- Always chooses the closet unvisited vertex
- Updates neighbors ("relaxing")
- Works until all shortest paths are Found
- Fails with negative weights

Date: __/__/20 Day: _____

DSA Bellman-Ford Algorithm

What it does:

- Finds the shortest path from one source vertex to all other vertices
- Works on directed graphs that may contain neg-edge weights
- Can also detect negative weight cycles, where no shortest path exists

Why Bellman-Ford (vs Dijkstra)?

- Dijkstra: Faster, but only works if all edges are non-negative.
- Bellman-Ford: Slower, but works even if there are negative edges and can detect negative cycles

How it works (Step-by-step)

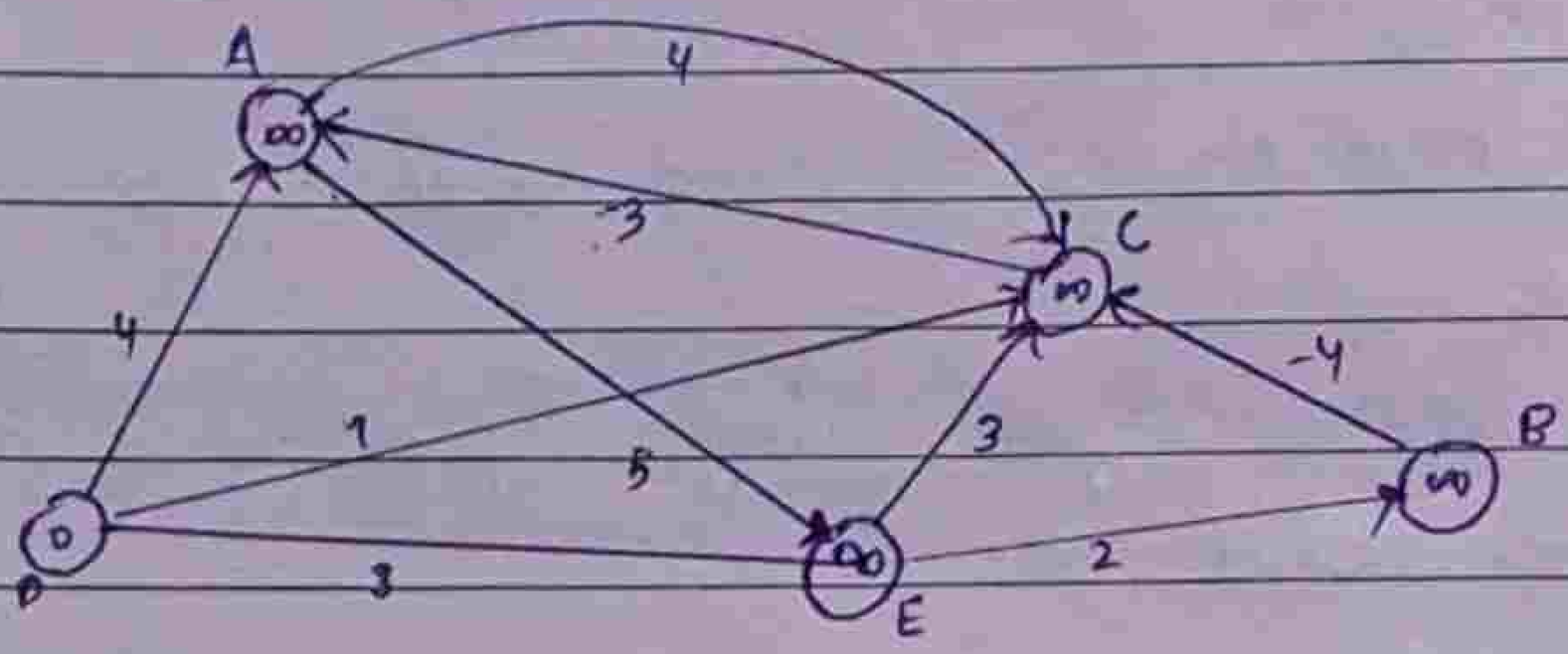
- Initialize distances:
  - Source Vertex = 0
  - All others = ∞ (Infinity)
- Relax all edges (V-1 times)
  - For every edge $u \rightarrow v$ with weight w:
    if distance[u] + w < distance[v], update distance[v]
  - This ensures shortest paths are found. (because the longest possible path without cycles has V-1 edge)
- Negative cycle detection (optional)
  - After V-1 rounds, check all edges again
  - if any distance can still be updated → negative cycle exists.

What is a Negative Cycle?

- A loop where total weight is negative

Date: __/__/20 Day: _____

- Example: A → C → A with weights adding upto -5
- You can keep going around and make the path "cheaper" forever → no shortest path exists



Example Idea:

- Source = D
- Distance update round by round when edges are relaxed.
- Paths like D → E → B → C → A can even give negative distances (if possible profitable edges exists (like fuel costs vs delivery payments)

Implementation (Main Ideas)

- Graph stored as adjacency matrix or edge list
- Relax edges V-1 times using nested loops
- keep track of predecessors to rebuilt actual path
- Extra loop for negative cycle detection.

Time Complexity

- Outer loop = O(V)
- Inner loop checks all edges = O(E)
- Total = O(V × E)
- Worst case (dense graph) = $O(V^3)$
- Slower than Dijkstra, but more powerful (handles negatives)

Date: __/__/20__ Day: _____

Summary

- Use Dijkstra if all weights ≥ 0 (faster)
- Use Bellman-Ford if negative edges exist or if you need to detect negative cycles
- It guarantees correct shortest paths in V-1 iteration
- If distances still change on the Vth iteration → negative cycle.

DSA Minimum Spanning Tree

- A tree formed from a graph that
  - Connects all vertices
  - Has no cycles (acyclic)
  - Has the minimum posible total edge weight
- Called a spanning tree because it "spans" all vertices
- Called minimum because the sum of edge weights is the smallest possible.

Real-World Example

- Imagine villages (vertices) that need electricity.
- Edges = possible cable routes with cost (distance, terrain, maintenance)
- MST = cheapest way to connect all villages with cable
- First MST algorithm (1926) was invented to connect villages in Moravia, Czeh Republic to the power grid

MST algorithms

- Prim's Algorithm
- Kruskal Algorithm

These are the two main famous algorithms to find MST

Date: __/__/20 Day: _____

Summary

- MST is unique if all edge weights are different
- If multiple edges have the same weight, there can be multiple MSTs (all with same cost)
- Prim's = grows like spreading electricity from one house to all.
- Kruskal's = sorts all possible roads, picks cheapest connection without cycles

OSA Prim's Algorithm

This is invented by Vojtech Jarnik (1930)
Rediscovered by prim (1957) & Dijkstra (1959)
Also called Jarnik's Algorithm or Prim-Jarnik Algorithm

What it does

Prim algorithm finds a Minimum Spanning Tree (MST) is a connected, undirected, weighted, graph.

- MST = a set of edge that connects all vertices with the minimum total edge weight.
- Works only if the graph is connected
- For disconnected graphs → use Kruskal's Algorithm

How it Works (step by step)

- Pick any starting vertex (random)
- From the currest MST, choose the smallest-weight edge going to vertex not yet in MST
- Add that edge and vertex to MST
- Repeat step 2-3 until all vertices are included.

Key Data Structures

- parents[] → Stores parents of each vertex (helps form MST tree structure)

Date: __/__/20 Day: _____

- in_mst[] → Tracks which vertices are already inside MST (avoid cycle)
- key_values[] → stores the shortest edge weight connecting a vertex to MST

Example (Manual Run, Start = A)

- Start with A
- Lowest edge is A-D (3) → add D
- Next lowest is A-B (4) or D-F (4) → pick B
- Next is B-C (3) → add C
- Then C-H (2) → add H
- Update edges → pick C-E (3) instead of B-E (6)
- Continue until all vertices (E, F, G) are added.

Python Implementation (Adjacency Matrix)

We use a Graph class with methods

- add_edge(u, v, weight) → adds undirected edge
- add_vertex_data(index, name) → names the vertex
- prims_algorithm() → runs Prim's algorithm using arrays

Time Complexity

Using arrays: $O(V^2)$

- Outer loop runs V times
- Each time, finding the min key takes O(V)
- Updating edges also takes O(V)

With priority Queue (Heap): $O(E \log V)$ → better for sparse graphs (few edges)

Date: / /20 Day:

DSA Kruskal's Algorithm

Find a Minimum Spanning Tree (MST) (or a Minimum Spanning Forest if the graph is disconnected)
MST = smallest set of edges that connects all vertices with minimum total weight, without cycles

Key idea
- Greedy algorithm → Always pick the smallest edge available that doesnot form a cycle.
- Cycle detection → Use Union-Find (Disjoint Set Union) (DSU) to check if adding an edge makes a cycle.
- MST size → for a connected graph with V vertices, MST will always have V → 1 edges.

Steps of Kruskal's Algorithm
- Sort all edges in non-decreasing order of weight
- Start with an empty MST
- For each edge (from smallest to largest weight)
  - if it connects two different components (no cycle) add it to MST
  - If it creates a cycle: skip it
- Stop when MST has V - 1 edges (or all edges checked if graph is disconnected)

Time Complexity
sorting edges = O(E log E)
Union-Find operations: O(1)
Total = O(E log E)

Date: __/__/20__ Day: _____

DSA Maximum Flow

The maximum flow problem is about Finding how much flow can be go through a directed graph, from a source vertex(S) to a sink vertex (t)
Each edge in the graph has:

- Capacity (c): the maximum possible flow on that edge
- Flow(f): how much flow is currently passing through

Example: an edge $v1 \rightarrow v_2$ with 0/7 means the current flow is 0, but up to 7 units can pas

Concepts

- Source (s): where the flow starts
- Sink(t): where the flow ends.
- Intermediate vertices: flow passes, through them.
- Conservation of flow: for all vertices (except s & t)
- Augmenting path: a path where more flow can be sent.
- Residual network: shows remaining capacity (capacity - current flow) and includes reverse edges or allow "sending back" flow.

Algorithms

- Ford-Fulkerson and Edmond's-Karp (gradually push more flow until no augmenting path remains
- They use the residual network to track unused and reversible capacity.

Multiple Sources and Sinks

If there are multiple sources or sinks, we create:

- A super-source(S): connects to all sources with infinite capacity.
- A super-sink (T): all sinks connect to it with

Date: __/__/20 Day: _____

infinite capacity.

- This allows Ford-Fulkerson / Edmonds-karp to work normaly

The Max-Flow Min-Cut Theorem

- A cut splits vertices into two sets:
  - Set S: includes the Source
  - Set T: includes the sink
- The cut capacity = sum of capacities of edges going from S → T
- The minimum cut is the cut with the smallest capacity (the bottleneck)
- The theorem states: maximum flow = capacity of min.cut

Example of cuts:

- Cut A: flow capacity > 14
- Cut B: capacity = 10
- Cut C: capacity = 8 → This is minimum cut, so the max. flow = 8

Practical Uses

- City planning: find where to add roads to aviod jams
- Utilities (water, electricity, networks): see where pipes/cable limit capacity.
- Manufacturing: Identify bottlenecks in production
- Logistics: improve supply chains by targeting weak links.

Mathematical Formulation

1- Capacity constraints:

$$f(u, v) \leq c(u, v)$$

Flow cannot exceed capacity

Date: __/__/20__ Day: _____

2- Anti- symmetry:

$$f(u,v) = -f(v,u)$$

Flow in one direction is negative in reverse

3- Conservation of flow: (except for s ant t)

$$\Sigma f(u,w) = 0$$

4. Total Flow

Flow leaving the source = Flow entering the sink

DSA Ford- Fulkerson Algorithm

The Ford-Fulkerson algorithm is used to solve Maximum Flow Problem in a directed graph

- Flow starts at a source vertex(s) and ends at a sink vertex(t).
- Each edge has a capacity (max flow allowed)
- The goal is to find the maximum possible flow from s to t

How it Works

- Start with 0 flow on all edges
- Find an augmening path(a path where more flow can be sent)
- Do a bottleneck calculation → find the minimum available capacity along that path.
- Increase flow on that path by bottleneck value
- Update residual network (remaining capacity)
- Repeat until no more augmenting paths are found.

Residual Network

- A graph showing the remaining capacities after

Date: __/__/20__ Day: _____

adding flow

- Residual capacity = capacity - current flow
- If v3 → v4 has capacity 3 and Flow 2 → residual = 1

Reverse Edges

- Ford - Flulkerson also adds reverse edges to allow sending flow back.
- Reverse edge capacity = current flow of forward edge
- Example: if v3 → v4 has flow 2, the reverse edge v4 → v3 has capacity 2
- This lets the algorithms "undo" flow if needed to increase total flow

Manual Run Example

- Start all flows = 0
- First augmenting path (DFS): s → v1 → v3 → v4 → t
  - Bottleneck = 3 → send flow = 3
- Next path: s → v2 → v1 → v4 → v3 → t (include reverse edge)
  - Bottleneck = 2 → send 2 units back & forward
- Next path: s → v2 → v1 → v4 → t
  - Bottleneck = 2
- Next path: s → v2 → v4 → t
  - Bottleneck = 1
- No more paths → Max Flow = 8

Conversation of flow

- Flow into any node (except s or t) = flow out of it

Time Complexity

O(V+E)

overall: O((V+E)·f)

for dense graphs (E > V) simplify to O(E·f)

Date: __/__/20 Day: _____

DSA Edmonds-Karp Algorithm

The Edmonds-Karp algorithm solves the maximum flow problem in a directed graph.

Why it's useful?

It's applied in network traffic optimization, manufacturing, supply chains, and airline scheduling

Key Idea.

- Flow goes from a source (s) to sink (t)
- Each edge has a capacity (max flow allowed)
- Edmonds-Karp is similar to Ford-Fulkerson, but it uses BFS (Breath-First Search) to find augmenting paths.

Steps of Algorithm

- Start with zero flow on all edges.
- Use BFS to find an augmenting path
- Find the bottleneck capacity (smallest residual capacity along the path)
- Send that much flow along the path.
- Repeat until no more augmenting paths exist

Residual Network

- Each edge has a residual capacity = capacity - current flow
- Shows how much more flow can still be pushed.
- Example: If flow = 2 and capacity = 3, residual = 1

Reverse Edges

- For every edge, a reverse edge is created

Date: __/__/20 Day: _____

- Reverse edge capacity = current flow of forward edge
- Allow pushing flow back if needed
- Example: If $v_1 \rightarrow v_3$ has flow 2, then residual capacity of $v_3 \rightarrow v_1$ = 2

Manual Walkthrough

- First BFS path: $s \rightarrow v_1 \rightarrow v_3 \rightarrow t$, bottleneck = 2 → flow sent = 2
- Next path: $s \rightarrow v_1 \rightarrow v_4 \rightarrow t$, bottleneck = 1, flow sent = 1
- Next path: $s \rightarrow v_2 \rightarrow v_4 \rightarrow t$, bottleneck = 3 → flow sent = 3
- Next path: $s \rightarrow v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow t$, bottlencet = 2 → flowsent = 2

At this point, no mere augmenting paths exist

- Maximum = flow = 8

Conversation of flow

- Flow into any intermediate vertex = flow out of it
- Flow out of Source (S) = Flow into sink (t)

Time Complexity

- Edmonds-Karp uses BFS ($O(E+V)$) to find path.
- In worst cases, BFS can run up to V.E times
- Total time Complexity = $O(V.E^2)$

Unlike Ford-Fulkerson, Edmonds-Karp does not depend on the max flow value, only one vertices and edges.

① DSA

Data Structure and algorithms.

What is efficiency in programming?

You can measure efficiency by 2 Things

- Time How much time it takes to run program
- Space How much space it takes to run program.

→ Do this with examples.

Google is a big example for time complexity.
It is the best search engine.
Google uses Page Rank algorithm. that is why google is so efficient.

→ Another example for space complexity.

For example you are the owner of facebook and you are going to hire a web developer.
For this you get Person A & Person B
CSS File For FB is.
A - candidite is saying 19kb & will charge 10Lac
B - candidate is saying 18Kb & will charge 30Lac.
Facebook daily active users are 2.11Billion

$200 \times 10^7 \times 1kb \Rightarrow 2 \times 10^9 \times 1kb$

$\Rightarrow 2 \times 10^3$ GB.

AWS - Per GB Bandwidth cost. is 10 rupees.
So user B - will save 73Lac & making so much Benefits.

→ Space efficiency matters.

Date: __/__/20__ Day: _____

What is Time Complexity?

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called Time complexity of the algorithm.

Techniques To measure Time Complexity.

There are 3 techniques

- Measuring time to execute
- Counting operations involved.
- Abstract notion of order of growth.

→ **Measuring Time (This is not good method)**

Print 1 to 100.

```
import time
start = time.time()
for in in range(1, 101):
    Print(i)
Print(time.time() - start)
```

* Time : 0.55 sec.

This is not a correct way of measuring time efficiency.

- Same code run on another machine it will give different result.
- If logic is same but implementation can reduce or increase the time.

For example you are using for loop now make a program using while loop It will effect the output.

Date: __/__/20__ Day: _____

```
Import time
start = time.time()
i = 1
while i < 101:
    Print(i)
    i += 1
Print (time.time() - start)
« output = 0.49 sec.
```

{Problems with these approach.}

1 Different time for different algo. ✓
2 Time varies if implementation changes ✓
3 Different machine diff time ✗
4 Does not work extremely small inputs ✗
5 Time varies for diff inputs But not make Relationship ✗

→ Counting Operations

- Assume these steps take constant Time
  - mathematical operations
  - comparisons
  - assignments
  - Accessing objs in memory
- Then count the number of operations excuted as Function of size of input.

```
def c_to_f(c):
    return c * 9.0/5 + 32
```

9 % 5 = 1 oper.
result x c = 2 oper.
result + 2 = 3 oper.
Total 3 operations.

Date: __/__/20__ Day: _____

```
def mysum (x):
    total = 0                    — (1)
    for i in range( x+1):        —(2)
        total += i               —(2)
    return total.
```

$1 + 3x = x = 10$ So 31

→ Problems with This Approach

Different time different Algo ✓

Time varies if implementation changes ✗

Different machine different time ✓

No clear defination of which oper to count ✗

Time varies for diff inputs, but can't ✓
establish a relationship

→ Order of Growth (Best in Industry)

what do we want

- we want to evaluate the algorithm
- we want to evaluate scalability
- we want to evaluate in term of input size

Different Inputs changes How Programs Run.

- a func that searches for an element in a list

```
def search_for_element(L, e):
    for i in L:
        if i == e:
            return true
    return False
```

when e is 1 element. Best case

when e in not in list Worst Case

when e is in middle Average Case

Date: __/__/20__ Day: _____

In Production we see worst case and This is every one use.

Order of Growth (Goals)

- Want to evaluate program efficiency when input is very big.
- Want to express the growth of Program run time as input size grows,
- want to put an opper bound on growth as tight as possible
- Do not need to be precise, order of not exact growth.
- We will look at largest factors in run time (which section of program will take longet)
- Thus we want tight upper bound on growth as function of size of input, in worst case.

Measuring order of growth: Big OH Notion

- Big oh notation measures an upper bound on the asymptotic growth, often called order of growth.

- Big oh or O() is used to describe worst case.
- Worst case occurs often and is the bottleneck when a programs run.
- Express rate of growth of program relative to the input size.
- Evaluate algorithm NOT machine or implementation.

Date: __/__/20 Day: ____

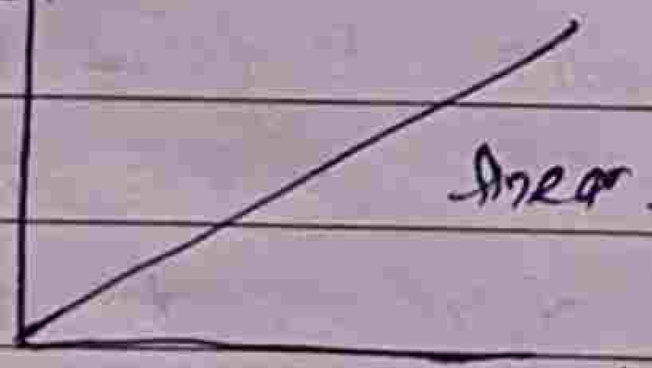
Exact Steps vs O()

```
def fact_inter(n):
    """ assumes n an Int >= 0 """
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

- computes Factoral
- numbers of steps.
  - worst case asymptotic complexity
  - Ignores additive constants
  - Ignores multiplicative constant.

$\underbrace{1}_{\text{input}} + 5n + 1$

$5n = n \Rightarrow O(n)$ — linear



Examples. → inside

$n^2 + 2n + 2$ → operation ~~inside loop~~ outside.

↓ nested loop → loop

$n^2 + n = n^2 \Rightarrow O(n^2)$ quadratic.

Date: __/__/20__ Day: _____

$n^2 + 10000n + 3^{1000}$ → constant.
ignored.

$n^2 = O(n^2)$

$Log(n) + n + 4$.
↓
small

$O(n)$

$0.0001 * n * log(n) + 300n$

$0.0001 * n * log(n)$

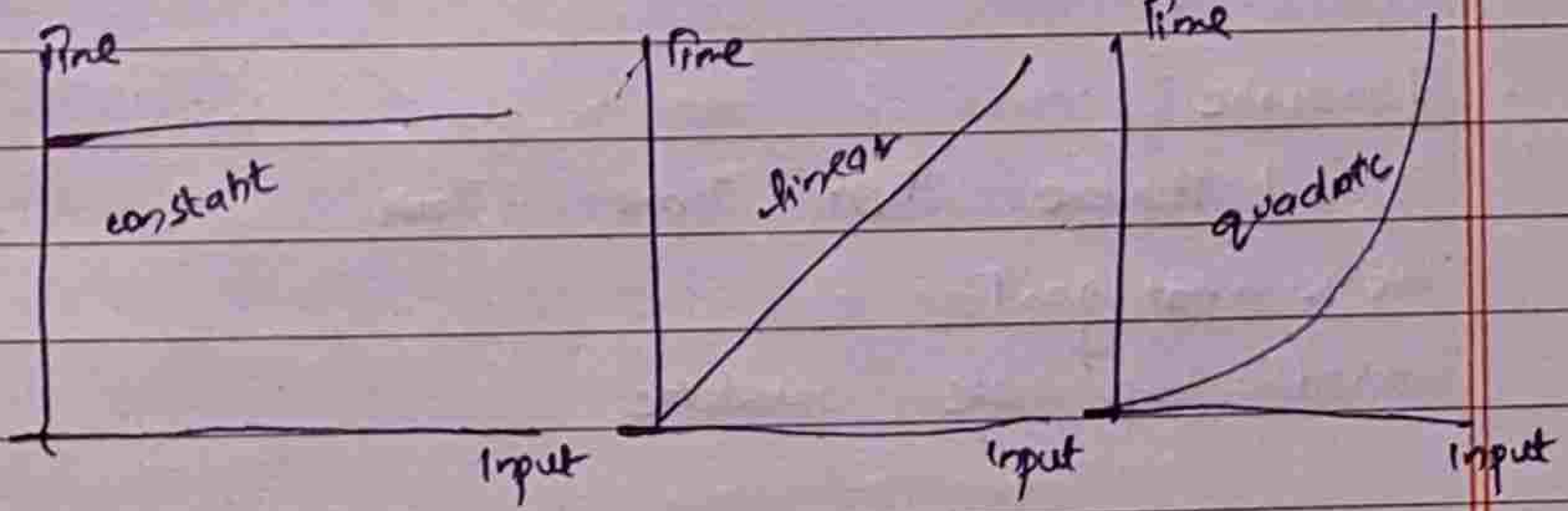
$n * log(n)$

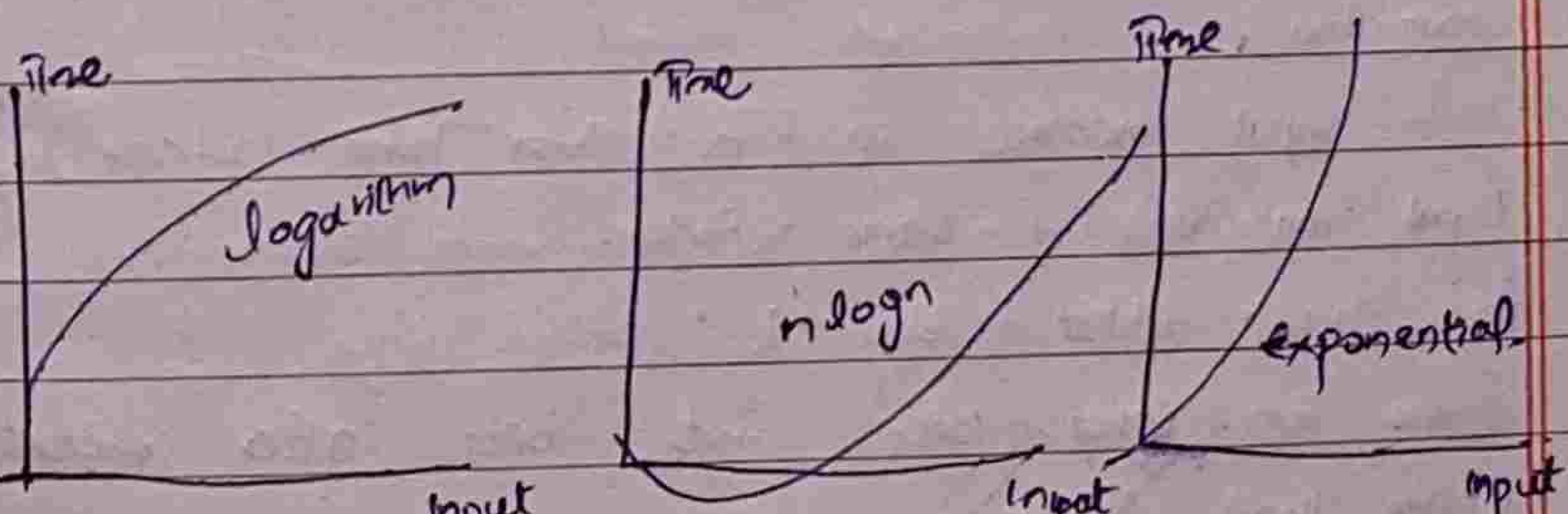
$nlog(n)$

$2n^{30} + 3n$

3^n

Types of order of growth.





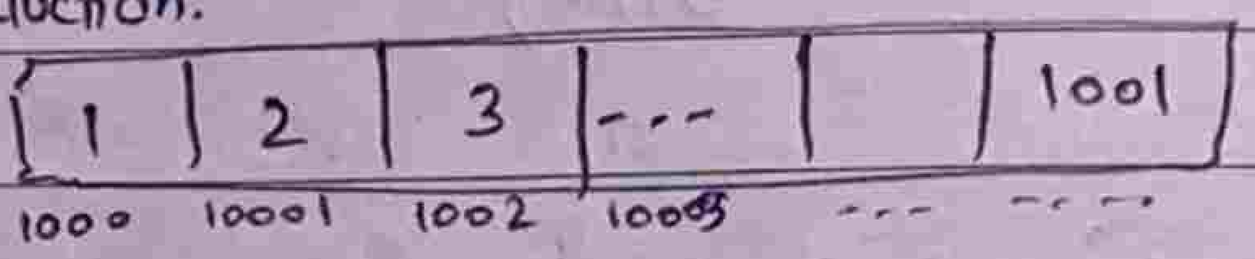
Date: / /20 Day:

- **Constant**

Constant doesnot depend on input.

Best for production.

len (A) =



A[100] = 75

A if len of A = 10000

A[100] = 75

if len of A = 100000

1st memory + 4(which is integer) x index.

O(1)

- **linear**

Time increase input increose

linear search, Array of length / Time.

This is good

- **Quadratic.**

Input double then Times 4 Times

This is not good.

Nested loop are quadratic.

- **Logarithm.**

If input increose 10 Times then time increose by 1

Input multiply by some Factor.

time increa added by 1

This is very best and best after constant

Better than linear.

Binary Search

- **n logn**

good than quadratic but not best than linear. / merge sort.

Sorting n > nlogn > n2

- **Exponential**

If input increase by 1
output/Time increases by 10
It is very bad and opposite of logarithm.

Complexity Growth Chart.

	Class	n = 10	= 100	= 1000	= 1000000
constant	O(1)	1	1	1	1
	O(logn) log.	1	2	3	6
linear	O(n) log(n)	10	100	1000	1000000
nlogn	O(n log n)	10	200	3000	6000000
quatratic	o(n^2)	100	10000	1,00000	10000000002
exponential	o(2^n)	1024	1234684 4568910 5678910	12345678 14101781012 16788 1011 11 10 9 8 7 6 5 4 3 1 12	infinity

Growth ↑

insult ↓

Date: __/__/20 Day: _____

DSA Euclidean Algorithm

The Euclidean Algorithm (named after the Greek mathematician Euclid, 300BCE) ins the oldest known algorithm

It is used to find the greatest common divisor (GCD) of two integers a and b.

The GCD is the largest number that divides both a and b without a remainder

For example: GCD(50, 15) = 15

How the Division Method words

The algorithm uses division with remainder

- Start with two numbers a and b
- Divide: $a = q_0 \cdot b + r_0$
- Replace a with b, and b with r_0
- Repeat until the reminder is 0.
- The last non-zero remainder is GCD

Manual Example: GCD (120, 25)

Step 1, 120 = 4·25 + 20

Step 2, 25 = 1·25 + 5

Step 3, 20 = 4·5 + 0

GCD (120, 25) = 25

Summary (key terminology)

- Divisor: Divides a number without remainder (6÷3=2)
- Remainder: What left after division (7÷3 = 2 rem 1)
- Common divisor: Divides both numbers (18 & 12 → 2, 3, 6)
- Greatest common divisor: largest common divisor (18, 12)=6

GCD is important in Number Theory & cryptography

Date: __/__/20 Day: _____

Implementation (Division Method)

```
def gcd_division(a, b):
    while b != 0:
        remainder = a % b
        print(f"{a} = {a//b} * {b} + {remainder}")
        a, b = b, remainder
    return a
print(gcd_division(120, 25))
```

The Original Euclidean Algorithm (Substraction Method)

Euclid's original method (2000+ years ago) used substraction instead of division.

Steps:

- Take numbers a and b
- Compute difference c = a - b
- Replace the largest number with difference
- Repeat until difference = 0
- The second last difference is GCD

Manual Example (substraction (GCD(120, 25))

120 - 25 = 95

95 - 25 = 70

70 - 25 = 45

45 - 25 = 20

25 - 20 = 5

20 - 5 = 15

15 - 5 = 10

10 - 5 = 5

5 - 5 = 0

GCD (120, 25) = 5

Date: __/__/20 Day: _____

Implementation (substraction Method)

```
def gcd_substraction(a, b):
    while a != b
        if a > b:
            print( f"{a} - {b} = {a-b}")
            a -= b
        else:
            print (f"{b} - {a} = {b-a}")
            b -= a
    return a
print(gcd_substraction(120, 25))
```

Comparsion: Division Vs Substraction

- Substraction Method: Repeatly substracts until no-match. Example: 25 is substracted 4 times from 120
- Division Method: Does the same in one step using remainder

Division is much Faster because one division = many substractions.

Both methods are based on sample principle GCD of a and b is also GCD of a-b and b

Summary

- Both methods always find GCD
- Substration is simple but slow
- Division is faster and preffered today.

Date: __/__/20 Day: _____

DSA Huffman Coding

Huffman Coding is an algorithm for lossless data compression. It is used in formats like ZIP, GZIP, PNG and even as part of lossy compression (MP3, JPEG)

It works by giving shorter codes to more frequent data and longer codes to less frequent data. No huffman code is a prefix of another, so decoding is always unambiguous.

How Huffman Coding Works

- Count frequencies: how often each character appears
- Built a binary tree: start from the least frequent nodes and merge them.
  - Left edge = 0, right edge = 1
  - Parent node's frequency = sum of its children
- Generate codes: follow edges from root to leaves to get each character's code.
- Encode text: replace characters with their huffman codes.

Example: Text = "lossless"

Step 1: Count characters

- s: 4
- l: 2
- o: 1
- e: 1

step2: Built huffman tree

- Combine o(1) and e(1) → parent 2
- Combine l(2) and above parent 2 → parent 4
- Combine that parent 4 and s(4) → root 8

Date: __/__/20 Day: _____

Step: Generate Codes

- S :→ 0
- l :→ 10
- o :→ 110
- e :→ 111

Step 4: Encode "lossless"

UTF-8 (8 bits per char)

01101110 01101111 01110011 01110011 01101100 01100101 01110011

(64 bits total)

Huffman Code

10 110 0 0 10 111 0

= 10110001011100 (14 bits)

Huge compression gain

Decoding Huffman Code

To decode, we need the conversion table

Example / Letter	Huffman Code
a	0
b	10
n	11

Decode 100110110

- 10 → b
- 0 → a
- 11 → n
- 0 → a
- 11 → n
- 0 → a

Result: banana

Date: __/__/20 Day: _____

DSA The traveling Salesman Problem

The travelling Salesman problem say:

A slalesman must visit several cities, visit each exactly once, and then reborn to the starting city.

Goal: Find the shortest possible route

Complexity of TSP

- To Find the true shortest route, we must check all possible routes.
- This has factoral time complexity O(n!)

Examples:

- 6 cities → 6! = 720 routes
- 8 cities → 8! = 40,320 routes
- 10 cities → 10! = 3.6+ million routes

Factorial (n!): e.g 4! = 4x3x2x1 = 24 possible route.

Time Complexity Summary

- Exact Solution (brute force): O(n!) → grows fast
- Greedy solution: $O(n^2)$ → much faster, but approx.

Optimization tricks:

- Fix starting city → reduces routes from n! to (n-1)!
- Since reversing a route gives the same distance only check half the routes → (n-1)/2
- Still overall O(n!), so only small n can be solved exactly.

Date: __/__/20__ Day: _____

DSA 0/1 Knapsack

You have a backpack with a weight limit and items (each with weight + value)

You must decide which items to take so that:

- Weight ≤ Capacity
- Value is maximized
- You can't take fractions (only 0 or 1 each item)

Real - world Uses

- Businesses: choose projects within a budget
- Logistics: decide which goods to load into trucks/Plane

Rules:

- Every item has weight + Value
- Bag has weight limit
- Pick items to maximize total value

Approaches

1. Brute Force
   - Check all combinations
   - Discard overweight ones
   - Pick the best value

How it works

- For each item: either include it (if capacity allows) or skip it.
- Recursively explore both
- Pick the best value Return the maximum

2. Memoization (Top-Down DP)
   - Store results of previous calls in a memo table
   - Avoid recalculating same stats (n, capacity)

Date: __/__/20 Day: _____

3. Tabulation (Bottom-up DP)

- Built a 2D table iteratively
- Row = items, Col = capacities
- For each cells:
  - If item fits → take max(include, exclude)
  - If too heavy → copy value from above row

Time Complexity:

- Brute Force: O(2^n) - very slow
- Memorization (Top-Down DP): o(n*C)
- Tabulation (Bottom up DP): o(n*C)

Dynamic programming works here because the problem has overlapping subproblems.

DSA Memomization

Memoization = storing results so we don't recompute the same thing again and again.
when used with recursion it is called Top-Down Dynamic Programming approaches (start with big problem → break into smaller ones)

Fibonacci Example using memoization
You can do this with recursion but plain recursion repeats work many times

```
def F(n):
    if memo[n] = None:
        return memo [n]
    print ('Computing F(' + str(n) + ')')
    if n <= 1
        memo[n] = n
    else:
```

Date: __/__/20__ Day: _____

```
        memo[n] = f(n-1) + f(n-2)
        return memo[n]
memo = [none] * 7
print ('F(6) = ', F(6))
print ('memo = ', memo)
```

Now only 7 computations instead of 25

For larger n, savings are huge

Big Difference

Finding F(30) →

- without memoization: ~2,692,537 computations
- with memoization: only 31 computations

DSA Tabulation

Tabulation = Solving problems using a table to store results of subproblems

- Start with smallest subproblems first, then built up until the final answer is found.
- This is called Bottom-Up DP approach
- Works only if the problem have overlapping sub problems.

Fibonannic with Tabulation

Tabulation fills a table starting from the base cases

- F(0) = 0, F(1) = 1
- Each next number:

f(x) = F(x-1) + F(x-2) x, or n

Find the 10th Fibonanci number:

```
def fibonanrci_tabulation(n):
    if n == 0: return 0
    if n == 1: return 1
```

Date: __/__/20 Day: _____

```
    F= [0] * (n+1)
    F[0], F[1] = 0, 1
    for i n range (2, n+1):
        F[i] = F[i-1] + f[i-2]
    print(F)
    return F[n]
n= 10
print(f"\n The {n}th Fibonacci no. is {fibonacci_tabulation(n)}")
```

Bottom up Vs Top-Down

- Tabulation (Bottom-up): starts at F(0), F(1), builds up step by step until F(10)
- Recursion / Memoization (Top-Down): starts from F(10) → keep breaking into smaller calls until reaching F(0) & F(1)

That is why tabulation is called bottom-up

Other Problems Solved with Tabulation

- 0/1 Knapsack Problem: choose items for max value within weight limit.
- Shortest path (Bellman-Ford): updates the "distance" array using tabulation
- Traveling Salesman (Held-Karp): use tabulation, faster than brute force but still heavy ($O(2^n * n^2)$).

Tabulation In DP

- Overlapping subproblems: same small problems solved multiple times. Example: F(3) overlaps with F(2) and F(1)
- Optimal substructure → solution can be built from smaller solutions. Example: $F(n) = F(n-1) + F(n-2)$

Both properties are needed to apply tabulation

Date: / /20 Day:

Dynammic Programming

Dynammic programming (DP) is a method to design algorithms by breaking a big problem into smaller ones, solving those subproblems, and combining their solutions to get the final answer.

For a problem to be solved with DP, it must have two properties

- Overlapping Subproblems: The problem can be split into smaller subproblems that are reused multiple times
- Optimal Substructure: The solution to the whole problem can be built from solutions to its sub-problems

Using DP to find nth Fibonacci number

The Fibonacci sequence starts with 0, 1 and every next number is the sum of the two before it

$$F(n) = F(n-1) + F(n-2)$$

The first 8 Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13

Here $F(4) = 3$

Steps to Design DP Algorithm

1- Check DP Properties

- Overlapping subproblems: Example: $F(6) = F(5) + F(4)$

Both $F(5)$ and $F(6)$ reuse $F(4)$

- Optimal substructure: Every Fibonacci number is built by adding the two before it

So, Fibonacci fits DP

Date: __/__/20 Day: _____

2- Solve the base cases

- F(0) = 0, F(1) = 1

3- Combine subproblem solutions

- Use the rule F(n) = F(n-1) + F(n-2) repeatly

4- Write the algorithm

- Example: To Find F(6), start with [0, 1], then keep adding until reaching index 6

```
def nth_fibo(n):
    if n == 0: return 0
    if n == 1: return 1
    F = [0] * (n + 1)
    F[0], F[1] = 0, 1
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
print(nth_fibo(6))
```

Techniques in Dynamic Programming

- Memoization (Top-Down)
  - Recursive approach
  - Stores results of solved subproblems to avoid recomputation
  - Called top-down because it starts from the main problem (F(n)) and breaks it into smaller ones.
- Tabulation (Bottom-Up):
  - Iterative approach
  - Stores solutions in table/array starting from base cases
  - Called bottom-up because it starts with small subproblems (F(0), F(1)) & builds up

Date: __/__/20__ Day: _____

Visual Difference:

- Tabulation (Bottom-up): Start from F(0) and go up step by step until F(10)
- Memoization (Top-Down): start from F(10) and break it down recursively into smaller problems until reaching F(0) and F(1)

Summary

Dynamic Programming = Solve overlapping subproblems once, store their results, and reuse them.

Techniques = Memoization (top-Down recursion) and Tabulation (bottom-Down iteration)

DSA Greedy Algorithms

A Greedy Algorithm makes decisions step by step, always choosing what looks best at the moment hoping this leads to the best overall result.

Example: Dijkstra Algorithm

In Dijkstra algorithm, the next vertex chosen is always the unvisited one with the shortest known distance from the source.

- This choice is greedy because it only uses current information, without considering the bigger picture

Greedy algorithms are one way to design solutions just like dyramic Programming (DP) is another.

Conditions For Greedy Algorithm

For a problem to be solved greedily, it must have

- Greedy Choice Property: The global optimum can

Date: __/__/20 Day: _____

can be reached by making locally optimal choices step by step

- Optimal Substructure - The overall best solution can be built from the best solutions of its subproblems

Example: of problems that fit:

- Sorting (selection Sort)
- Shortest Path (Dijkstra Algorithm)

Problems that do not fit:

- Traveling salesman Problem (TSP)
- 0/1 Knapsack Problem

The 0/1 Knapsack Problem

Rules:

- Each item has weight + value
- Knapsack has a weight limit
- You can only take whole items (not fractions)

Goal: Maximize total value

Why Greedy Fails:

- Example: Weight Limit = 10kg
  - Shield: 5kg, $300
  - Pot: 4kg, $500
  - Horse: 7kg, $600

Greedy choice (take the horse, most valueable) gives only $600 Better choice (shield + Pot) give $800

The Traveling Salesman Problem (TSP)

Rules: Visit each city once, return to start

Goal: Find the shortest total route

Date: / / 20 Day:

Why Greedy Fails:

- A greedy method (always visit the nearset unvisited city) does not guarantee the shortest overall path
- Routes may cross unnecessarily, increasing distance
- TSP does not satisfy greedy properties.

Fact: There is no efficient algorithm for exact TSP
The only way is to try all routes → Time Comp O(n!)
Greedy can give a decent approximation, but not the optimal solution

Summary

- Greedy = Local choices → Global solution (works only if problem has Greedy Choice Property + Optimal Substructure)
- Works For: Sorting, Dijkstra's
- Fails For: 0/1 Knapsack, TSP
- Not all problems can be solved greedily, — Sometimes we need DP or other methods

The End

"Thanks For Reading"

written by Mirza Yasir Abdullah Baig