DIPLOMA WALLAH

(Your **One Stop Hub** For Diploma Resources)

DATA STRUCTURES

WITH PYTHON

Complete Notes Based on Full Syllabus

- Diploma Engineering

4th Semester

www.diplomawallah.in

Notes prepared by **Sangam**

Diploma Wallah Sangam

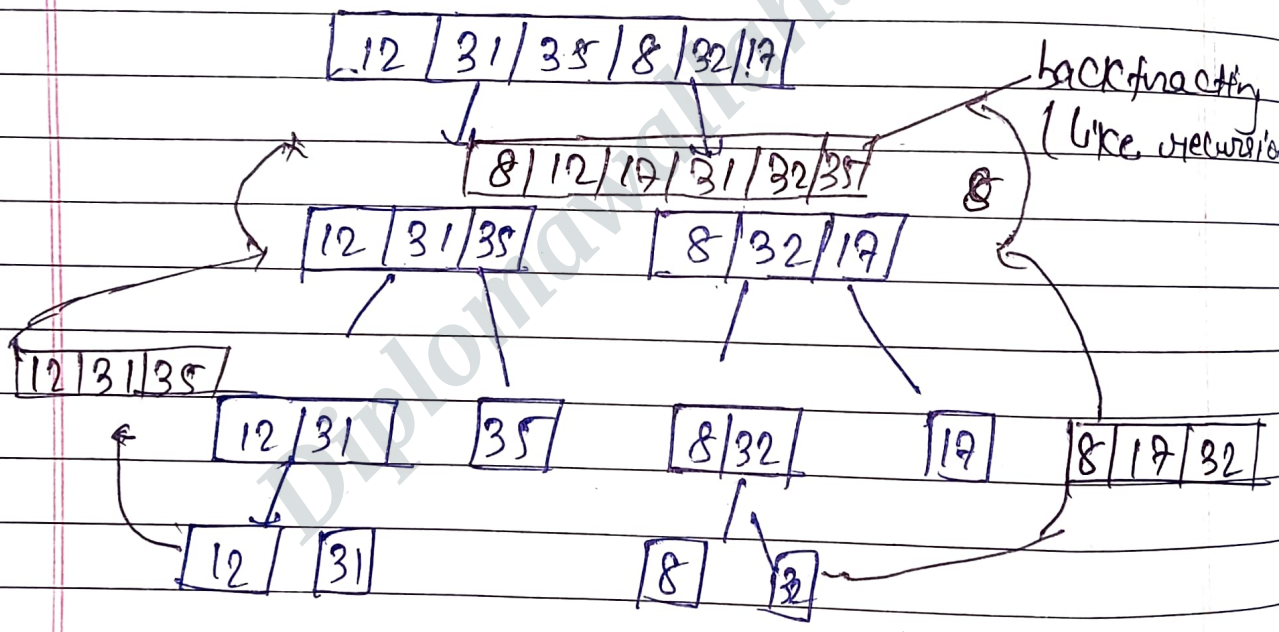
unit - 04

Divide and Conquer

* Merge Sort :-

(i) Divide and Conquer.

arr[] = {12, 31, 35, 8, 32, 17}



(i) Divide the array (by mid)

(ii) merge parts to create a sorted array

①

Recursive function : (Dividing part)

①

```
void merge_sort(arr[], start, end)
{
    if(start<end)
    {
        int mid = start + (end-start)/2

        mergesort(arr, start, mid)      // left side
        mergesort(arr, mid+1, end)      // right side.

        merge(arr, start, mid, end)
    }
}
```

s=0 end=3

| 12 | 31 | 35 | 8 |

mid = 0 + (3-0)/2 = 1

// left mergesort.

| 12 | 31 |

s=0 e=1

mid = (s+e)/2 = (0+1)/2 = 1

| 12 |

s=0 e=0

| 31 |

s=e=1

Equal ho gya means isse ek phle ka loop chalega.

// right

| 35 | 8 |

s=2 e=3

mid = (2+3)/2 = 2

| 35 |

s=e=2

| 8 |

s=e=3

(11) merge sort · (2) part · other programming

12	31	35

8	17	32

```
void merge (arr, start, mid, end)
{
    merge = int >temp;
    i = start, j = mid+1;
    while (i<=mid && j<=end)
    {
        if (A[i] <= [A[j])   ← left side
            temp.pb (arr[i])
            i++;   ↳ pushback.
        else{
            temp.pb (A[j])  ← right side
            j++
        }
    }

for idx = 0; idx < temp . idx++)
A[idx + start] = temp[idx]

}
```

Python code: —

```
def merge_sort(arr):
    if len(arr) <= 1  # means if arr is smaller
        return arr        than 1 then return arr.

    mid = len(arr)//2
# Divide step                    start
    left_half = merge_sort(arr[:mid])   (mid-1)
    right_half = merge_sort(arr[mid:])   end
# both divide the array into smaller
                part.
# merge steps.
    return merge_sort(left_half, right_half)

def merge(left, right):
    result = []
    i=j=0  → starting mai dono apna 0 index se start hoga
# merge while comparing element
    while i<len(left) and j<len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i = i+1
        else:
            result.append(right[j])
            j=j+1
```

Recursion se divide karte hain.
Sorted parts ko merge karte hain.
Space thoda lagta hai but time efficient hai.

Add remaining element after merge.

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

Example usage

```
arr = [6,3,9,5,2,8]
sorted_arr = merge_sort(arr)
print("Sorted array is", sorted_arr)
```

O/P → Sorted array is [2,3,5,6,8,9]

Dry run

Step 1: Divide

[6,3,9] [5,2,8]
→ [6] [3,9] [5] [2,8]
→ [6], [3], [9] [5] [2], [8]

Step 2: Merge

→ [3,9]] first left
→ [3,6,9]]
→ [2,8]] right
→ [2,5,8]]

Step 3: Final merge

[2,3,5,6,8,9]

Merge Sort is a popular sorting algorithm known for its time efficiency and stability (O(log n).

It follow the divide and conquer approach to sort at a given array of elements.

Here's a Step:—

1. Divide:— Divide the list of array recursively into two halves until it can no more be divided.
2. Conquer:— Each subarray is sorted individually using the merge sort algorithm.
3. Merge:— The sorted subarrays are merged back together in sorted order.
   The process continues until all elements from both subarrays have been merged.

Time complexity -

Best case = O(n log n)

Average case = O(n log n)

Best case = O(n log n)

* Quick Sort :-

working in 3 steps :-

(i) pick the pivot

(ii) partition → choose the pivot index.

(iii) QS (left) → (i) pick the pivot, partition (until 1 element not left)

QS (right) "

Ex | 5 | 2 | 6 | 4 | 1 | 3 |

let 3 = pivot

↳ pivot element are kept in middle and if the other element is less than pivot the stone in left half or if highest then stone in right half.

| 2 | 1 | (3) | 5 | 6 | (4) — pivot

(1)

| 1 | 2 | & | 4 | 5 | 6 |

↳

(6)

4 & | 5 | 6 |

that is index space.

(1) pick the pivot

```
quick_sort(arr, start, end)
if(start<end)
{
    pivot_index = partition(arr, start, end)
    quicksort(arr, start, pivot_index-1)
    quicksort(arr, pivot+1, end)
}
```

pivot element index ← pivot_index = partition(arr, start, end)
left ← quicksort(arr, start, pivot_index-1)
right ← quicksort(arr, pivot+1, end)

(11) partition

```
partition(a, start, end)
    idx = start-1; (Means it takes negative
    pivot = arr[end]; extra space to store
    for(j=start; j<end; j++)          number.)
    {
        idx = idx+1;
        swap(arr[j], arr[idx])
    }
}
idx = idx+1
swap(arr[end], arr[idx])
return idx;
}
```

Python code

```
def quick_sort(arr):
    if len(arr)<=1
        return arr
    pivot = arr[-1] #last element ko pivot lena.
    left=[]
    right=[]
#partition using loop.
    for i in range(len(arr-1)): #pivot koi aur value
        if arr[i]<=pivot:
            left.append(arr[i])
        else:
            right.append(arr[i])
#Recursively sort & combine.
    return quicksort(left)+[pivot]+
            quick_sort(right)
#Example
arr=[38,27,43,3,9,82,10]
print("Original array", arr)
Sorted_arr = quick_sort(arr)
print("Sorted array:", Sorted_arr)
```

o/p → Original array [38,27,43,3,9,82,10]
Sorted array: [3,9,10,27,38,43,82]

(11) Binary Search :-

Time Comp. $\log(n)$

0	1	2	3	4
10	20	30	40	50

→ The array must be in sorted form.

```
def binary(array, target)
    left, right = 0, len(array)-1
    while left <= right:
        mid = (left + right)//2
        if array[mid] == target
            return mid
        elif array[mid] < target
            left = mid + 1
        else
            right = mid - 1
    return -1
array = [10, 20, 30, 40, 50]
target = 40
result = binary(array, target)
if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

Binary Search is a searching algorithm used to find the position of a target element within a sorted array.
It works on the principle of divide and conquer by repeatedly dividing the search interval into half.

Code working :
1. Start with two pointers :
   - low - first index of the array
   - high - last index of the array

2. Find the middle index.
   mid = (low + high) / 2
   - if target == arr[mid] - element found
   - if target < arr[mid] - search in the left half (high = mid - 1)
   - If target > arr[mid] - search in the right half (low = mid + 1)
   - Repeat steps until the element is found or the range becomes empty.

* Time Complexity
  - Best case : O(1) - Element found at first mid
  - Average / worst case : O(log n) → each step halves the search space.
* Space complexity

- Iterative : O(1) → No extra memory
- Recursive : O(log n) - due to function call stack.

Advantage
- much faster than linear search for large datasets.
- Efficient for sorted data.

Disadvantage
- Works only on sorted arrays.
- Requires random access (works well on arrays, not linked list).

* Dynamic programming

Dynamic Programming (DP) is a method to solve problems by breaking them into smaller subproblems, solving each subproblems once, and storing the result for future use (memoization or tabulation).
It avoides recomputing the same results, which makes algorithms must faster.
(Time overlapping subproblems ko eg ki baar solve karke unka result store kar liya jata hai, taki future me dubara calculation na karni pade).

Ex- Fibonacci Sequence.

Formula:-

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

$F(0) = 0, F(1) = 1$

Why Dynamic prog. (DP) for fibonacci?

- Recursive method me same problem bar-bar solve hote hain → Time complexity = $O(2^n)$ (very slow)
- Dynamic programming me hum subproblem ko result ek array me store karte hain → Time complexity = $O(n)$.

Approach to solve.

1. Recursive Approach - Solve by using formula
2. DP → Top down (memoization) - recursion ke sath results ko store karke reuse karna
3. DP → Bottom up (Tabulation) - Chhote value se start karke array fill karna.
4. Space optimized Approach - Sirf last 2 value ko store karna.

(Har approach ka time complexity or space down alag hota hai).

Approach in DP

a) Top-down (memoization)

- Recursive approach + Storing result in a dict/list.

```
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n<=1:
        return n
    memo[n] = fibonacci_memo(n-1, memo) +
    fibonacci_memo(n-2, memo)
        return memo[n]
print(fibonacci_memo(7))  #output: 13
```

b) Bottom-up (Tabulation)

- Iteratively calculate from base case up ton.

```
def fibonacci_tab(n):
    dp = [0] * (n+1)  # Create table of size n+1
    dp[0], dp[1] = 0, 1  # Base case
    for i in range(2, n+1):  # Fill from 2 to n
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
print(fibonacci_tab(7))      #output: 13
```

dry run:

i	dp[i-1]	dp[i-2]	dp[i]
0	-	-	0
1	-	-	1
2	1	0	1

Storing [0, 1, 0, 0, 0, 0, 0]

i	dp[i] = dp[i-1] + dp[i-2]	
2	dp[1] + dp[0]	// [0, 1, 1 ...]
3	dp[2] + dp[1]	[0, 1, 1]
4	dp[3] + dp[2]	// [0, 1, 1, 2, 3 ...]
5	dp[4] + dp[3]	// [0, 1, 1, 2, 3, 5]
6	dp[5] + dp[4]	// [0, 1, 1, 2, 3, 5, 8]
7	dp[6] + dp[5]	// [0, 1, 1, 2, 3, 5, 8, 13]
8	~~dp[7] + dp[6]~~	

dp[1]
↳ independent.

Time Complexity : O(n)
Space " : O(n) for tabulation,
O(1) if only last two value stored.

Diploma Wallah

Date
Page

*Backtracking

Backtracking is an algorithm technique used to solve problem by building a soln. step by step and abandoning (backtracking) a path as soon as it determined that it cannot lead to a valid soln.

- Try → Check → Explore → undo
- make a choice → if valid, move forward → if not, go back and try another choice.
- Implemented mainly using recursion.

Ex 1. Start walking through a maze.
2. If you hit a dead end, go back to the last junction.
3. Try a new path until you find the exit.

General pseudocode:-

```
function backtrack (position):
    if solution is complete:
        record/print solution
        return
    for each choice in possible choices:
        if choice is valid:
            make choice
            backtrack (next position)
            undo choice  //backtrack.
```

maze problem

```
def is_safe(board, row, col, n):
    # check column
    for i in range(row):
        if board[i][col] == 1:
            return false.
    # Check upper-left diagonal
    for i, j in zip(range(row-1, -1, -1), range
        (col-1, -1, -1)):
        if board[i][j] == 1:
            return false.
    # check upper-right diagonal
    for i, j in zip(range(row-1, -1, -1),
            range(col+1, n)):
        if board[i][j] == 1:
            return false
    return true
def solve(board, row, n):
    # if all Question are placed
    if row >= n:
        for r in board:
            print(r)
        print()
        return
```

# Try placing queen in all column of this row.
for col in range(n):
    if is_safe(board, row, col, n):
        board[row][col] = 1 # place queen.
        solve(board, row+1, n) # Recursion call
        board[row][col] = 0 # backtrack

n = 4
board = [[0]*n for _ in range(n)]
solve(board, 0, n)

Steps Working
1. Start row 0 se, har column me queen rakh ke try karte hain.
2. Agar safe hai → place queen → next row ke liye recursion call.
3. Agar kahin aage problem aayi → queen hatao (backtrack) → next column try karo.
4. Jab sab rows fill ho jaye → ek valid arrangement print hota hai.

- is_safe() - Check karta hai ki queen ka position safe hai ya nai.
- board[row][col] = 0 → yahi backtracking ka undo step hai
- Recursion ke andar hi har possibility try hoti hai.