

|  |  |
| --- | --- |
| **SL.** | **Problem Description** |
| 01 | Write a program to sort a linear array using the bubble sort algorithm. |
| 02 | Write a program to find an element using the linear search algorithm. |
| 03 | Write a program to sort a linear array using the merge sort algorithm. |
| 04 | Write a program to find an element using the binary search algorithm. |
| 05 | write a program to find a given pattern from text using the pattern matching algorithm. |
| 06 | write a program to implement a queue data structure along with it's typical operations. |
| 07 | Write a program to solve n queen's problem using backtracking. |
| 08 | Consider a set S = {5, 10,12, 13, 15, 18) and d = 30. Write a program to solve the sum of subset problem. |
| 09 | Write a program to solve the following 0/1 Knapsack using dynamic programming approach profits P = (15,25,13,23), weight W = (2,6,12,9), Knapsack C = 20, and the number of items n=4. |
| 10 | Write a program to solve the Tower of Hanoi problem for the N disk. |

**Problem No.1**

**Problem Name:** Sorting a Linear Array Using the Bubble Sort Algorithm.

**Title:** Sorting a Linear Array Using Bubble Sort Algorithm.

**Theory:** Sorting is a fundamental operation in computer science, enabling efficient organization of data for searching, analysis and storage. The Bubble Sort algorithm is one of the simplest sorting techniques. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until no swaps are needed, indicating the list is sorted.

The name “Bubble sort” comes from the way smaller elements “bubble” to the top of the array with each pass. Although its simplicity makes it easy to understand and implement, Bubble Sort is inefficient for large datasets due to its 0(n^2) time complexity in the worst and average cases.

**Algorithm:**

1.Start array size of n.

2.Initialize a flag (swapped) as False.

3.For each pass:

* Iterate through the array from the first element to n-i-1 where i is the current pass index.
* Compare each pair of adjacent elements.
* If the current element is greater than the next, swap them and set swapped=True.

4. If no elements were swapped in a pass, terminate the loop(the array is sorted).

5. Repeat the process for n-1 passes or until the array is sorted.

6.Output the sorted array.

7. End.

**Source Code:**

def bubble\_sort(arr):

n = len(arr)

for i in range(n - 1):

swapped = False

print(f"Pass {i + 1}:")

for j in range(n - i - 1):

print(f" Comparing {arr[j]} and {arr[j + 1]}", end="")

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

print(" -> Swapped to:", arr)

swapped = True

else:

print(" -> No swap")

print(f" Array after pass {i + 1}: {arr}")

if not swapped:

break

arr = [5, 3, 8, 6, 2]

print("Array before sorting:", arr)

bubble\_sort(arr)

print("Array after sorting:", arr)

**Output:**

Array before sorting: [5, 3, 8, 6, 2]

Pass 1:

Comparing 5 and 3 -> Swapped to: [3, 5, 8, 6, 2]

Comparing 5 and 8 -> No swap

Comparing 5 and 3 -> Swapped to: [3, 5, 8, 6, 2]

Comparing 5 and 8 -> No swap

Comparing 5 and 8 -> No swap

Comparing 8 and 6 -> Swapped to: [3, 5, 6, 8, 2]

Comparing 8 and 6 -> Swapped to: [3, 5, 6, 8, 2]

Comparing 8 and 2 -> Swapped to: [3, 5, 6, 2, 8]

Comparing 8 and 2 -> Swapped to: [3, 5, 6, 2, 8]

Array after pass 1: [3, 5, 6, 2, 8]

Array after pass 1: [3, 5, 6, 2, 8]

Pass 2:

Comparing 3 and 5 -> No swap

Pass 2:

Comparing 3 and 5 -> No swap

Comparing 3 and 5 -> No swap

Comparing 5 and 6 -> No swap

Comparing 6 and 2 -> Swapped to: [3, 5, 2, 6, 8]

Array after pass 2: [3, 5, 2, 6, 8]

Comparing 6 and 2 -> Swapped to: [3, 5, 2, 6, 8]

Array after pass 2: [3, 5, 2, 6, 8]

Array after pass 2: [3, 5, 2, 6, 8]

Pass 3:

Pass 3:

Comparing 3 and 5 -> No swap

Comparing 5 and 2 -> Swapped to: [3, 2, 5, 6, 8]

Array after pass 3: [3, 2, 5, 6, 8]

Pass 4:

Array after pass 3: [3, 2, 5, 6, 8]

Pass 4:

Comparing 3 and 2 -> Swapped to: [2, 3, 5, 6, 8]

Comparing 3 and 2 -> Swapped to: [2, 3, 5, 6, 8]

Array after pass 4: [2, 3, 5, 6, 8]

Array after sorting: [2, 3, 5, 6, 8]

**Problem No:2**

**Problem name:** Write a program to find an element using a linear search algorithm.

**Title:** Sorting a Linear Array Using linear search Algorithm.

**Theory:** Linear Search is one of the simplest search algorithms used to find the position of a target element in a list. It involves iterating through the list element by element, and comparing each element with the target.

Time Complexity:

Best Case: 0(1) when the target is the first element.

Worst Case: 0(n) when the target is the last element or not present.

Space Complexity: 0(1), as it does not require additional memory.

**Algorithm:**

Linear Search ( A, N, item)

1.Loc=-1

2.i=1

3.Repeat while I ≤ N and A[I]≠ item

I=I+1

4.if A [I]=item

Loc=I

5. Return Loc

**Source Code:**

def linear\_search(arr, target):

print("Pass-by-pass Execution:\n")

for i in range(len(arr)):

print(f"Pass {i+1}: Checking index {i}, Value = {arr[i]}")

if arr[i] == target:

print(f"\nElement {target} found at index {i}.\n")

return i

print(f"\nElement {target} not found in the array.\n")

return -1

if \_\_name\_\_ == "\_\_main\_\_":

arr = [10, 20, 30, 40, 50, 60]

target = 40

result = linear\_search(arr, target)

**Output:**

Pass-by-pass Execution:

Pass 1: Checking index 0, Value = 10

Pass 2: Checking index 1, Value = 20

Pass 3: Checking index 2, Value = 30

Pass 4: Checking index 3, Value = 40

Element 40 found at index 3.

**Problem No.3**

**Problem Name:** Sorting a Linear Array Using the Merge Sort Algorithm.

#### **Title:** Sorting a Linear Array Using the Merge Sort Algorithm

Theory: Merge Sort is a Divide and conquer algorithm thet efficiently sorts an array by recursively dividing it into smaller subarrays,sorting these subarrays, and then merging them back together. This algorithm is more efficient than simpler algorithms like Bubble Sort for large datasets, eith a time complexity of 0( n log n) in the worst, average, and best cases.

**Algorithm:**

1.Base Case Check. If length(array)>1 then: Check if the array has more than one element.

2. Divide the Array. Mid= length(array)//2:Find the middle index of the array.

3. Recursive Calls. MergeSort ( leftHalf): Recursively sort the left half of the array.

4. Initialize Indices.

5. Merge the Halves.

6.Copy Remaining Elements.

**Source Code:**

def merge\_sort(arr, level=0):

if len(arr) > 1:

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

print(f"{' ' \* level}Splitting: {arr}")

merge\_sort(left\_half, level + 1)

merge\_sort(right\_half, level + 1)

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i] < right\_half[j]:

arr[k] = left\_half[i]

i += 1

else:

arr[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

arr[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

arr[k] = right\_half[j]

j += 1

k += 1

print(f"{' ' \* level}Merging: {arr}")

if \_\_name\_\_ == "\_\_main\_\_":

arr = [38, 27, 43, 3, 9, 82, 10]

print("Initial Array:", arr)

merge\_sort(arr)

print("Sorted Array:", arr)

**Output:**

Initial Array: [38, 27, 43, 3, 9, 82, 10]

Splitting: [38, 27, 43, 3, 9, 82, 10]

Splitting: [38, 27, 43]

Splitting: [27, 43]

Merging: [27, 43]

Merging: [27, 38, 43]

Splitting: [38, 27, 43, 3, 9, 82, 10]

Splitting: [38, 27, 43]

Splitting: [27, 43]

Merging: [27, 43]

Merging: [27, 38, 43]

Splitting: [38, 27, 43]

Splitting: [27, 43]

Merging: [27, 43]

Merging: [27, 38, 43]

Splitting: [27, 43]

Merging: [27, 43]

Merging: [27, 38, 43]

Merging: [27, 43]

Merging: [27, 38, 43]

Merging: [27, 38, 43]

Splitting: [3, 9, 82, 10]

Splitting: [3, 9]

Merging: [3, 9]

Splitting: [82, 10]

Merging: [10, 82]

Merging: [3, 9, 10, 82]

Merging: [3, 9, 10, 27, 38, 43, 82]

Sorted Array: [3, 9, 10, 27, 38, 43, 82]

**Problem No.4**

**Problem Name:** Write a program to find an element using the binary search algorithm.

Title: Finding an Element Using the Binary Search Algorithm

**Theory:** Binary Search is a highly efficient algorithm for finding the position of a target element within a stored array. It works by repeatedly dividing the search interval in half. If the target element matches the middle element, the search is successful. Otherwise, depending on whether the target is smaller or larger than the middle element, the search continues in the left or right half of the array, respectively.

This method is significantly faster than linear search for large datasets, with a time complexity of 0( log n).

**Algorithm:**

Binary\_Search(A,N,item)

1.LOC=-1

2. B=1, E=N

3. while B≤E

4. mid=[(B+E)/2]

5.if item=A[mid] then

LOC=mid[Exit Loop]

Else if item>A[mid]

B=mid+1

Else

E=mid-1

6. Return LOC

**Source Code:**

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

pass\_num = 1

while low <= high:

mid = (low + high) // 2

print(f"Pass {pass\_num}: low = {low}, high = {high}, mid = {mid}, arr[mid] = {arr[mid]}")

if arr[mid] == target:

print(f"\nElement {target} found at index {mid}.\n")

return mid

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

pass\_num += 1

print(f"\nElement {target} not found in the array.\n")

return -1

if \_\_name\_\_ == "\_\_main\_\_":

arr = [10, 20, 30, 40, 50, 60, 70, 80, 90]

target = 70

result = binary\_search(arr, target)

**Output:**

Pass 1: low = 0, high = 8, mid = 4, arr[mid] = 50

Pass 2: low = 5, high = 8, mid = 6, arr[mid] = 70

Element 70 found at index 6.

**Problem No.5**

**Problem Name:** Write a program to find a given pattern from text using the pattern matching algorithm.

**Title**: Finding a Given Pattern from Text Using a Pattern Matching Algorithm.

**Theory:** Pattern matching is a computional process used to find occurences of a specific pattern ( a substring) within a large text. It is widely used in ext processing, search engines, and DNA sequence analysis. The Naïve Pattern Matching Algorithm is one of the simplest methods to achieve this. It compares the pattern with all possible substrings of the text, one by one, to identify matches.

**Algorithm:**

1. Let the text have n characters and the pattern have m characters.
2. For each position I in the text from 0 to n-m:

Compare the substring of length m string at I with the pattern.

If all characters match, record the position I as a match.

1. If no matches are found, return an appropriate message.
2. Output all string indices of the pattern in the text.

**Source Code:**

def naive\_pattern\_match(text,pattern):

n=len(text)

m=len(pattern)

match\_indices=[]

for i in range(n-m+1):

if text[i:i+m]==pattern:

match\_indices.append(i)

return match\_indices

if \_\_name\_\_=="\_\_main\_\_":

text="ABABABCABABABCABABABC"

pattern="ABABC"

print("Text: ", text)

print("PAttern: ", pattern)

matches=naive\_pattern\_match(text,pattern)

if matches:

print(f"Pattern found indices at: {matches}")

else:

print("Pattern Not found")

**Output:**

Text: ABABABCABABABCABABABC

PAttern: ABABC

Pattern found indices at: [2, 9, 16]

**Problem No.6**

**Problem Name:** Write a program to implement a queue data structure along with it’s typical operations.

**Title:** Implementation of a Queue Data Structure and Its Typical

Operations.

**Theory:** A Queue is a linear data structure that follows the First In, First Out (FIFO) principle. Elements are added to the rear (enqueue operation) and removed from the front (dequeue operation). Queues are widely used in scenarios like task scheduling, breadth-first search in graphs, and buffering data streams.

**Algorithm:**

1. Start
2. Check if the queue is full.
3. If the queue is full, produce an overflow error and exit.
4. If the queue is not full, increment the rear pointer to point to the next space.
5. Add a data element to the queue location, where the rear is pointing.
6. End the process and exit.

**Source Code:**

class Queue:

def \_\_init\_\_(self):

self.items = []

def enqueue(self, item):

self.items.append(item)

print(f"Enqueued: {item}")

def dequeue(self):

if self.is\_empty():

print("Queue is empty. Cannot dequeue.")

return None

return self.items.pop(0)

def peek(self):

if self.is\_empty():

print("Queue is empty. Nothing to peek.")

return None

return self.items[0]

def is\_empty(self):

return len(self.items) == 0

def display(self):

print(f"Queue: {self.items}")

if \_\_name\_\_ == "\_\_main\_\_":

queue = Queue()

queue.enqueue(10)

queue.enqueue(20)

queue.enqueue(30)

queue.display()

print(f"Dequeued: {queue.dequeue()}")

print(f"Front element: {queue.peek()}")

print(f"Is queue empty? {queue.is\_empty()}")

queue.display()

**Output:**

Enqueued: 10

Enqueued: 20

Enqueued: 30

Queue: [10, 20, 30]

Dequeued: 10

Front element: 20

Is queue empty? False

Queue: [20, 30]

**Problem No 07:**

**Problem Name:** Write a program to solve n queen's problem using backtracking.

**Title:** Solving the N-Queens Problem Using Backtracking.

**Theory:** TheN-Queens problemis a classic combinatorial problem that involves placing N queens on an N × N chessboard such that no two queens attack each other. This means:

* No two queens share the same row.
* No two queens share the same column.
* No two queens share the same diagonal.

To solve this problem efficiently, we use backtracking, which explores all possible placements and backtracks when an invalid placement is found.

**Algorithm:**

1. Start with an empty board of size N × N.
2. Place a queen in the first available row in a way that it does not attack any previously placed queens.
3. Move to the next row and repeat the process.
4. If a valid position is found in all N rows, print the solution.
5. If a row has no valid placement, backtrack to the previous row and try the next possible position.
6. Repeat until all solutions are found.

**Source Code:**

def print\_board(board):

for row in board:

print(" ".join("Q" if col else "." for col in row))

print("\n")

def is\_safe(board, row, col, n):

for i in range(row):

if board[i][col]:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j]:

return False

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j]:

return False

return True

def solve\_n\_queens(board, row, n):

if row == n:

print\_board(board)

return True

for col in range(n):

if is\_safe(board, row, col, n):

board[row][col] = 1

solve\_n\_queens(board, row + 1, n)

board[row][col] = 0

def n\_queens(n):

board = [[0] \* n for \_ in range(n)]

solve\_n\_queens(board, 0, n)

if \_\_name\_\_ == "\_\_main\_\_":

N = 4

print(f"Solutions for {N}-Queens Problem:\n")

n\_queens(N)

**Output:** Solutions for 4-Queens Problem:

. Q . .

. . . Q

Q . . .

. . Q .

. . Q .

Q . . .

. . . Q

. Q . .

**Problem No:08**

**Problem Name:** Consider a set S = {5, 10,12, 13, 15, 18) and d = 30. Write a program to solve the sum of subset problem.

**Title:** Sum of Subset Problem Using Backtracking.

## **Theory:** The Sum of Subset Problem is a well-known combinatorial problem in which we need to find all subsets of a given set S whose elements sum up to a given value d.

## **Algorithm:**

1. Sort the set S (optional but improves efficiency).
2. Start with an empty subset and a sum of 0.
3. For each element, either:
   1. Include it in the subset and move to the next element.
   2. Exclude it and move to the next element.
4. If the sum of selected elements equals d, print the subset.
5. If the sum exceeds d or no more elements are left, backtrack.
6. Repeat until all subsets are checked.

**Source Code:**

def sum\_of\_subsets(S, subset, index, total, target):

if total == target:

print(subset)

return

if total > target or index >= len(S):

return

sum\_of\_subsets(S, subset + [S[index]], index + 1, total + S[index], target)

sum\_of\_subsets(S, subset, index + 1, total, target)

if \_\_name\_\_ == "\_\_main\_\_":

S = [5, 10, 12, 13, 15, 18]

d = 30

print(f"Subsets of {S} that sum up to {d}:\n")

sum\_of\_subsets(S, [], 0, 0, d)

**Output:**

Subsets of [5, 10, 12, 13, 15, 18] that sum up to 30:

[5, 10, 15]

[5, 12, 13]

[12, 18]

**Problem No: 09**

**Problem Name:** Write a program to solve the following 0/1 Knapsack using dynamic programming approach profits P = (15,25,13,23), weight W = (2,6,12,9), Knapsack C = 20, and the number of items n=4.

**Title:** 0/1 Knapsack Problem Using Dynamic Programming.

## **Theory:** The 0/1 Knapsack Problem is a classic optimization problem in which we are given a set of n items, each with:

* Profit (P[i])
* Weight (W[i])

We have a knapsack of capacity C, and we must determine the maximum profit that can be obtained by selecting items without exceeding the capacity. Each item can either be included (1) or not (0), hence the name 0/1 Knapsack.

## **Algorithm:**

1. Initialize a DP table dp[n+1][C+1] with zeros.
2. Iterate over each item i (1 to n).
3. For each capacity w (1 to C):
   1. If W[i-1] > w, exclude the item.
   2. Otherwise, decide whether to include or exclude it using the max profit formula.
4. Return the final maximum profit from dp[n][C].

**Source Code:**

def knapsack(P, W, C, n):

dp = [[0] \* (C + 1) for \_ in range(n + 1)]

for i in range(1, n + 1):

for w in range(1, C + 1):

if W[i - 1] <= w:

dp[i][w] = max(dp[i - 1][w], P[i - 1] + dp[i - 1][w - W[i - 1]])

else:

dp[i][w] = dp[i - 1][w]

return dp[n][C], dp

def print\_selected\_items(dp, W, P, C, n):

w = C

selected\_items = []

for i in range(n, 0, -1):

if dp[i][w] != dp[i - 1][w]:

selected\_items.append((P[i - 1], W[i - 1]))

w -= W[i - 1]

print("\nSelected Items (Profit, Weight):", selected\_items)

if \_\_name\_\_ == "\_\_main\_\_":

P = [15, 25, 13, 23]

W = [2, 6, 12, 9]

C = 20

n = 4

max\_profit, dp\_table = knapsack(P, W, C, n)

print(f"Maximum Profit: {max\_profit}")

print\_selected\_items(dp\_table, W, P, C, n)

**Output:**

Maximum Profit: 63

Selected Items (Profit, Weight): [(23, 9), (25, 6), (15, 2)]

**Problem No:10**

**Problem Name:** Write a program to solve the Tower of Hanoi problem for the N disk.

**Title:** Tower of Hanoi Problem Using Recursion.

## **Theory:** The Tower of Hanoi is a classic recursive problem in which we have three rods (A, B, C) and N disks of different sizes. The objective is to move all the disks from the source rod (A) to the destination rod (C) using the auxiliary rod (B), following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or on an empty rod.
3. All disks must be moved while following the above constraints.

This problem demonstrates the power of recursion, as the solution involves solving smaller subproblems.

## **Algorithm:**

1. Base Case: If there is only one disk, move it directly from A to C.
2. Recursive Case:
   1. Move N-1 disks from A to B using C as an auxiliary rod.
   2. Move the Nth (largest) disk directly from A to C.
   3. Move the N-1 disks from B to C using A as an auxiliary rod.
3. Repeat until all disks are moved from A to C.

The minimum number of moves required to solve the problem for N disks is given by the formula:

*M(N)=2N−1M(N) = 2^N - 1*M(N)=2N−1

where N is the number of disks.

**Source Code:**

def tower\_of\_hanoi(n, source, destination, auxiliary):

if n == 1:

print(f"Move disk 1 from {source} to {destination}")

return

tower\_of\_hanoi(n - 1, source, auxiliary, destination)

print(f"Move disk {n} from {source} to {destination}")

tower\_of\_hanoi(n - 1, auxiliary, destination, source)

if \_\_name\_\_ == "\_\_main\_\_":

N = int(input("Enter the number of disks: "))

print(f"\nSteps to solve Tower of Hanoi for {N} disks:\n")

tower\_of\_hanoi(N, 'A', 'C', 'B')

Output:

Steps to solve Tower of Hanoi for 4 disks:

Move disk 1 from A to B

Move disk 2 from A to C

Move disk 1 from B to C

Move disk 3 from A to B

Move disk 1 from C to A

Move disk 2 from C to B

Move disk 1 from A to B

Move disk 4 from A to C

Move disk 1 from B to C

Move disk 2 from B to A

Move disk 1 from C to A

Move disk 3 from B to C

Move disk 1 from A to B

Move disk 2 from A to C

Move disk 1 from B to C