

INDEX

|  |  |
| --- | --- |
| **Sl.** | **Problem Statement** |
| **1.** | |  | | --- | | Write a program to sort a linear array using the bubble sort algorithm. | |
| **2.** | |  | | --- | | Write a program to find an element using a linear search algorithm. | |
| **3.** | |  | | --- | | Write a program to sort a linear array using the merge sort algorithm. | |
| **4.** | |  | | --- | | Write a program to find an element using the binary search algorithm. | |
| **5.** | |  | | --- | | Write a program to find a given pattern from text using the pattern matching algorithm. | |
| **6.** | |  | | --- | | Write a program to implement a queue data structure along with its typical operations. | |
| **7.** | |  | | --- | | Write a program to solve **n** queen's problem using backtracking. | |
| **8.** | |  | | --- | | Consider a set **S = {5,10,12,13,15,18}** and **d = 30**. Write a program to solve the sum of subset problem. | |
| **9.** | |  | | --- | | 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 01: Write a program to sort a linear array using the bubble sort algorithm.**

**Introduction:** Sorting enables the arrangement of elements in a specific sequence, either ascending or descending. The Bubble Sort Algorithm is one of the most basic and widely recognized sorting methods.

Bubble Sort functions by consistently comparing adjacent items in an array and exchanging them if they are out of order. This procedure goes on until the full array is arranged. The algorithm derives its name from how smaller elements "rise" to their appropriate positions with every pass. While it may not be the most efficient sorting technique, it is a great tool for grasping the fundamentals of sorting principles and algorithmic reasoning.

**Algorithm:** The Bubble Sort algorithm operates through the following process:

Step 1: Begin with an unsorted array of sizes.

Step-2: Execute the subsequent for n-1:

Traverse the array from the first element to the penultimate element.

Examine each adjacent pair of elements.

Exchange their positions if they are not in the correct order (i.e. if the first element is larger than the second for sorting in ascending order).

If no exchanges are made, conclude the process early, as the array is already arranged.

Step 3: The array will be sorted upon completing all required iterations.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source code:**

#bubble\_sort arr

**def** bubble\_sort**(**arr**):**

n**=len(**arr**)**

**for** i **in** **range(**n**):**

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

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

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

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

**return** arr

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

arr**=[**0**,**50**,**60**,**40**,**13**,**56**,**32**,**9**,**123**]**

**print(**"Original array:"**,**arr**)**

sorted\_arr**=**bubble\_sort**(**arr**)**

**print(**"Sorted array:"**,**sorted\_arr**)**

**Sample Input and Output:**

* Input: arr = [64, 34, 25, 12, 22, 11, 90]
* Output: Original arr: [64, 34, 25, 12, 22, 11, 90]

Pass 1: [34, 25, 12, 22, 11, 64, 90]

Pass 2: [25, 12, 22, 11, 34, 64, 90]

Pass 3: [12, 22, 25, 11, 34, 64, 90]

Pass 4: [12, 11, 22, 25, 34, 64, 90]

Pass 5: [11, 12, 22, 25, 34, 64, 90]

Pass 6: [11, 12, 22, 25, 34, 64, 90]

Pass 7: [11, 12, 22, 25, 34, 64, 90]

Sorted array: [11, 12, 22, 25, 34, 64, 90]

**Problem 02: Write a program to find an element using a linear search algorithm**.

**Introduction**: Linear Search is a simple algorithm that examines each item in a list one by one until it either finds the target element or runs out of items. While this method is easy to apply, it tends to be efficient for large datasets compared to more sophisticated search algorithms.

**Algorithm:**

1. Begin with the first item in the list

2. Compare the current item to the target value

3. If the current item corresponds with the target return of its index

4. If it does not match, proceed to the next item

5. Continue this process until the item is located or the end of the list is reached.

6. If the item cannot be found, return -1

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source code:**

**def** linear\_search**(**arr**,** target**):**

n **=** **len(**arr**)**

**for** i **in** **range(**n**):**

**if** arr**[**i**]** **==** target**:**

**return** i

**return** **-**1

arr **=** **[**12**,** 67**,** 90**,** 34**,** 50**]**

target **=** 50

**print(**"Original array:"**,** arr**)**

**print(**"Target:"**,** target**)**

index **=** linear\_search**(**arr**,** target**)**

**if** index **!=** **-**1**:**

**print(**f"Element {target} found at index {index}"**)**

**else:**

**print(**f"Element {target} not found"**)**

**Sample Input and Output:**

~Input:

* + - arr: [34, 78, 12, 89, 45, 23]
    - Target **:** 89

~Output:

* Original array: [34, 78, 12, 89, 45, 23]
* Target:50
* Element 89 found at index: 3

**Problem03: Write a program to sort a linear array using the merge sort algorithm.**

**Introduction:** Merge Sort is a sorting algorithm that uses the divide-and-conquer approach. It splits the array into two halves sorts each segment separately, and then combines the sorted segments to create a fully sorted array. This algorithm has an average and worst-case time complexity of O(n log n), which makes it more efficient for larger when compared to others such as Bubble Sort or Insertion Sort.

**Merge Sort Steps:**

**~Divide:** Split the array into two equal parts.

**~Conquer**: Sort each part recursively.

**~Merge:** Combine the two sorted parts into one sorted array

**Algorithm:**

**1. Divide:**

~If the array contains one or no elements, return it as it is.

~Determine the middle index: mid = len(arr) // 2.

~Break the array into two segments: left\_half = arr[:mid] and right\_half = arr[mid:].

**2. Conquer:**

Recursively sort both segments:

~left\_half = merge\_sort(left\_half)

~right\_half = merge\_sort(right\_half)

3. Merge:

~Set up pointers for both segments and create an empty list to hold the merged result.

~While there are still elements in both segments:

~Compare the elements and add the smaller one to the merged list.

~Add any leftover elements from both segments to the merged list.

4. Return:

Return the sorted, merged list.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source Code:**

# Merge Sort Implementation

**def** merge\_sort**(**arr**):**

n **=** **len(**arr**)**

**if** n **<=** 1**:**

**return** arr

mid **=** n **//** 2

l\_half **=** arr**[:**mid**]**

r\_half **=** arr**[**mid**:]**

# Recursively sort both halves

l\_half **=** merge\_sort**(**l\_half**)**

r\_half **=** merge\_sort**(**r\_half**)**

**return** merge**(**l\_half**,** r\_half**)**

**def** merge**(**left**,** right**):**

new **=** **[]**

i **=** j **=** 0

# Merge elements from left and right into new in sorted order

**while** i **<** **len(**left**)** **and** j **<** **len(**right**):**

**if** left**[**i**]** **<** right**[**j**]:**

new**.**append**(**left**[**i**])**

i **+=** 1

**else:**

new**.**append**(**right**[**j**])**

j **+=** 1

new**.**extend**(**left**[**i**:])**

new**.**extend**(**right**[**j**:])**

**return** new

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

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

**print(**"Original Array:"**,** arr**)**

sorted\_arr **=** merge\_sort**(**arr**)**

**print(**"Sorted Array:"**,** sorted\_arr**)**

**Sample Input and Output:**

~Input:

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

~Output:

Original Array: [60, 20, 40, 10, 90, 30, 80, 50]

Sorted Array: [10, 20, 30, 40, 50, 60, 80, 90]

**Problem 04: Write a Program to find an element using the binary search algorithm.**

**Introduction:** The binary search algorithm is a very effective way to search sorted arrays. To discover a target element, it drastically cuts down on the number of comparisons required by repeatedly halving the search interval. Binary search is significantly faster than linear search techniques for bigger datasets because of its O(log n) time complexity.

**Algorithm:**

1. Initialization:  
  
~Two pointers should be set:

* low to the initial index (0).
* high to the array's final index (length - 1).

2. Repetition Although low is equivalent to or less than high:  
  
~Use the formula mid = (low + high) // 2 to determine the middle index.  
~Examine the target value with relation to the middle element (arr[mid]):  
~Return mid (target found) if arr[mid] equals the target.  
~Set low to mid + 1 (look in the right half) if arr[mid] is below the goal.  
~Set high to mid-1 if arr[mid] exceeds the goal (see on the left side).  
  
3. Return -1 if the desired value cannot be located.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source code:**

#binary\_search

**def** binary\_search**(**arr**,**target**):**

n**=len(**arr**)**

left**=**0

right**=**n**-**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

#input

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

target**=**5

**print(**"Original Array:"**,**arr**)**

**print(**"Target:"**,**target**)**

index**=**binary\_search**(**arr**,**target**)**

**if** index**!=-**1**:**

**print(**f"Element{target}found at index{index}"**)**

**else:**

**print(**f"Element{target} not found"**)**

**Sample Input and Output:**

~Input:

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

target = 5

~Output:

Original Array: [10, 20, 30, 40, 50]

Target: 5

Element 5 not found

**Problem 05: Write a program to find a given pattern from text using the pattern-matching algorithm.**

Introduction: The purpose of pattern matching, a basic computer science activity, is to locate instances of a given pattern (substring) in a longer text string. A popular technique for this purpose is the Knuth-Morris-Pratt (KMP) algorithm, which effectively locates every instance of a pattern in a text. The temporal complexity of the KMP algorithm is O(n + m), where n is the text's length and m is the pattern's.

**KMP Algorithm Steps**

1. **Preprocess the Pattern:**
   * Create the longest prefix-suffix (LPS) array for the pattern, which stores the length of the longest proper prefix that is also a suffix for each prefix of the pattern.
2. **Search:**
   * Use the LPS array to skip unnecessary comparisons when a mismatch occurs, allowing the search to proceed efficiently through the text.

**Algorithm:**

1. **Preprocess the Pattern:**

* Create the Longest Prefix Suffix (LPS) array:
  + Initialize lps array of the same length as the pattern.
  + Set length to 0 and iterate through the pattern with index i.
  + If pattern[i] matches pattern[length], increment both length and i, and set lps[i].
  + On mismatch, update length to lps[length - 1] if length is not zero; otherwise, set lps[i] to 0.

2. **Search for the Pattern in the Text:**

* Initialize pointers i (text) and j (pattern) at 0.
* While i is less than the text length:
  + If text[i] matches pattern[j], increment both i and j.
  + If j equals the pattern length, print the match index i - j, and reset j using lps[j - 1].
  + On mismatch, if j is not zero, update j to lps[j - 1]; else, increment i.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source Code:**

#naive\_pattern\_matching

**def** naive\_pattern\_matching**(**text**,**pattern**):**

n**=len(**text**)**

m**=len(**pattern**)**

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

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

**print(**f"Pattern found at index{i}"**)**

**return** **True**

**print(**"Pattern not found"**)**

**return** **False**

text**=**"aaabbba"

pattern**=**"aaaa"

naive\_pattern\_matching**(**text**,**pattern**)**

**Sample Input and Output**:

~Input:

text = "aaabbba"

pattern = "aaaa"

naive\_pattern\_matching(text, pattern)

~Output:

Pattern not found

**Problem 06: Write a program to implement a queue data structure along with its typical operations.**

**Introduction:** The First In First Out (FIFO) principle governs the linear data structure known as a queue. The first element added to a queue is also the first element withdrawn. Queues are frequently utilized in many different applications, including scheduling, graph breadth-first search, and service system request handling.

Operations in Queues  
1. Enqueue: Fill the queue with an element.  
2. Dequeue: Take out and give back the queue's front element.  
3. Peek: Put the front element back in place without taking it out.  
4.IsEmpty: Verify if there is nothing in the queue.  
5. Display: Make every item in the queue visible.

**Algorithm:**

1. Start  
2. Establish the class `Queue`  
3. Within the class `Queue`:   
  
a) Create a blank list called `items`.   
To add {item` to `items`, define `enqueue(item)`.   
c) Use `dequeue()` as defined:   
"Queue is empty" should be returned if the queue is empty.   
Otherwise, take out and send back the first item.   
d) Create `peek()`: - Return "Queue is empty" if the queue is empty.   
Otherwise, send back the first item.   
e) Use `is\_empty()` as defined:   
If the queue is empty, return True; otherwise, return False.   
g) Define `size()`: - Returns the number of queue entries   
  
4. Finish

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source Code:**

**class** **Queue:**

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

self**.**items **=** **[]**

**def** enqueue**(**self**,** item**):**

self**.**items**.**append**(**item**)**

**def** dequeue**(**self**):**

**if** **not** self**.**items**:**

**return** "Queue is empty"

**return** self**.**items**.**pop**(**0**)**

**def** peek**(**self**):**

**if** **not** self**.**items**:**

**return** "Queue is empty"

**return** self**.**items**[**0**]**

**def** is\_empty**(**self**):**

**return** **len(**self**.**items**)** **==** 0

**def** size**(**self**):**

**return** **len(**self**.**items**)**

queue **=** Queue**()**

queue**.**enqueue**(**10**)**

queue**.**enqueue**(**20**)**

**print(**queue**.**dequeue**())**

**print(**queue**.**peek**())**

**print(**queue**.**is\_empty**())**

**print(**queue**.**size**())**

**Sample Input and Output:**

~Input :

queue = Queue()

queue.enqueue(10)

queue.enqueue(20)

print(queue.dequeue())

print(queue.peek())

print(queue.is\_empty())

print(queue.size())

~Output:

10

20

False

1

**Problem 07: Write a program to solve n queen’s problem using backtracking.**

**Introduction:** A famous combinatorial problem, the N-Queens problem involves arranging N queens on a N×N chessboard in such a way that no two queens share a row, column, or diagonal. A backtracking strategy, which methodically investigates every configuration and gets rid of invalid placements early, can be an effective way to handle the issue.

**Algorithm:**

**Algorithm** The backtracking approach works as follows:

1. Place a queen in the first available row and column.
2. Move to the next row and try placing another queen in a safe position.
3. If a safe position is found, proceed to the next row; otherwise, backtrack and try a different column.
4. Continue this process until all queens are placed successfully.
5. If a solution is found, print the board; otherwise, report failure.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source Code:**

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

**for** i **in** **range(**row**):**

**if** board**[**i**]** **==** col **or** \

board**[**i**]** **-** i **==** col **-** row **or**

board**[**i**]** **+** i **==** col **+** row**:**

**return** **False**

**return** **True**

**def** solve\_n\_queen**(**n**,** board**=None,** row**=**0**):**

**if** board **is** **None:**

board **=** **[-**1**]** **\*** n

**if** row **==** n**:**

print\_board**(**board**,** n**)**

**return**

**for** col **in** **range(**n**):**

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

board**[**row**]** **=** col

solve\_n\_queen**(**n**,** board**,** row **+** 1**)**

board**[**row**]** **=** **-**1

**def** print\_board**(**board**,** n**):**

**for** i **in** **range(**n**):**

line **=** **[**'.'**]** **\*** n

line**[**board**[**i**]]** **=** 'Q'

**print(**' '**.**join**(**line**))**

**print(**"\n"**)**

# Run for N = 8

solve\_n\_queen**(**8**)**

**Sample Input and Output:**

~Input:

solve\_n\_queen(8)

~Output:

. . . . . . Q .

. . . . Q . . .

. . . Q . . . .

Q . . . . . . .

. . . . . Q . .

. . Q . . . . .

. . . . . . . Q

. Q . . . . . .

. . . Q . . . .

. . . . . Q . .

. Q . . . . . .

. . . . . . Q .

Q . . . . . . .

. . . . . . . Q

. . Q . . . . .

. . . . Q . . .

. . . . Q . . .

. . . . . . Q .

. . Q . . . . .

. . . . . . . Q

Q . . . . . . .

. . . Q . . . .

. . . . . Q . .

. Q . . . . . .

. . . . . . . Q

Q . . . . . . .

. . . Q . . . .

. . . . . Q . .

. . Q . . . . .

. . . . . . Q .

. Q . . . . . .

. . . . Q . . .

Q . . . . . . .

. . . . . Q . .

. . Q . . . . .

. . . Q . . . .

. . . . . . . Q

. . . . Q . . .

. Q . . . . . .

. . . . . . Q .

**Problem 08: Consider a set S={5,10,12,13,15,18} and d=30.Write a program to solve the sum of subset problem.**

**Introduction**: We concentrate on a particular example of the subset S={5,10,12,13,15,18} and a goal total d=30 in this lab report. Finding every potential subset of S that adds up to d is the aim. This exercise emphasizes the intricacy of combinatorial issues while simultaneously reinforcing knowledge of recursion and backtracking techniques.  
  
We can identify which subsets satisfy the requirement of reaching the intended sum by methodically examining the combinations of members inside the set. This paper describes the approach used, how the algorithm was implemented in Python, and the outcomes of the program's execution. We learn more about the computational difficulties of the subset sum problem and the effectiveness of recursive methods in resolving it as a result of this investigation.

**Algorithm:**

1. **Input:** Set Sand target sum d.

2.**Function Call:** Start with find\_subsets\_with\_sum(S, d, [], 0).

3.**Base Cases:**

* If target\_sum == 0: Print the current subset.
* If index >= len(S) or target\_sum < 0: Return.

4. **Recursive Cases:**

* **Include the Current Element:**
  + Call find\_subsets\_with\_sum(S, target\_sum - S[index], current\_subset + [S[index]], index + 1).
* **Exclude the Current Element:**
  + Call find\_subsets\_with\_sum(S, target\_sum, current\_subset, index + 1).

1. **Backtrack:** Return to explore other elements.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source Code:**

**from** itertools **import** combinations

S **=** **list(map(int,** **input(**"Enter the elements separated by spaces: "**).**split**()))**

d **=** **int(input(**"Target of the sum: "**))**

**print(**"Subsets with sum equal to"**,** d**,** ":"**)**

found **=** **False**

**for** r **in** **range(**1**,** **len(**S**)** **+** 1**):**

**for** subset **in** combinations**(**S**,** r**):**

**if** **sum(**subset**)** **==** d**:**

**print(list(**subset**))**

found **=** **True**

**if** **not** found**:**

**print(**"Subset not found"**)**

**Sample Input and Output:**

~Input:

Enter the elements separated by spaces: 1 2 3 4 5

Target of the sum: 5

~Output:

Subsets with sum equal to 5 :

[1, 4]

[2, 3]

[5]

**Problem 09: Write a program to solve the following 0/1 knapsack using dynamic programming appraoch profit P=(15,25,13,23),weight W=(2,6,12,9),Knapsack C=20,and the number of items n=4 .**

**Introduction:** A classic algorithmic problem in operations research and computer science is the 0/1 Knapsack problem. Finding the number of each item to include in a collection given a set of objects, each with a weight and a profit, is the aim in order to maximize the overall profit while keeping the total weight below or equal to a specified limit.

**Algorithm:**

1.Enter the following information: Read the profits array P[, weights array W[, knapsack capacity C, and item count n.  
  
2. Initialise: Make a 2D array called dp with the dimensions (n+1) x (C+1) and set its initial value to 0.  
  
3. Loop through Items: Each item from 1 to n is looped through.  
  
4. Iterate through Capacities: Repeat steps 1 through C for every item.  
  
5. Verify Weight: Update the dp value if W[i-1] is less than or equal to the current capacity w.  
  
6. To determine the maximum, let dp[i][w] = max(dp[i-1][w], P[i-1] + dp[i-1][w-W[i-1]]).  
  
7. Exclude Item: Set dp[i][w] = dp[i-1][w] if W[i-1] is higher than w.  
  
8. Output: Determine the greatest profit by returning dp[n][C].

**TAHSINA SABA Roll:220622 Session:2021-22**

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

**else:**

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

**return** dp**[**n**][**C**]**

n **=** **int(input(**"Enter number of items: "**))**

P **=** **list(map(int,** **input(**"Enter profits separated by space: "**).**split**()))**

W **=** **list(map(int,** **input(**"Enter weights separated by space: "**).**split**()))**

C **=** **int(input(**"Enter knapsack capacity: "**))**

# Solve knapsack

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

**print(**"Maximum Profit:"**,** max\_profit

**Sample Input and Output:**

~Input:

Profits (P): (15, 25, 13, 23)

Weights (W): (2, 6, 12, 9)

Knapsack Capacity (C): 20

Number of Items (n): 4

~Output: Maximum Profit: 40

**Problem 10: Write a program to solve the Tower of Hanoi problem for the N disk**.

**Introduction:** Three rods and several disks of varying sizes that slot onto any rod make up the Tower of Hanoi, a mathematical puzzle. The disks are arranged on a single rod in ascending order of size at the beginning of the puzzle, with the smallest disk at the top. Moving the complete stack to a different rod while adhering to the following guidelines is the aim:  
  
1. You can only move one disk at a time.  
2. The upper disk is taken from one of the stacks and placed on top of an empty rod or another stack after each move.  
3. A smaller disk cannot be stacked on top of another disk.

**Algorithm:**

1. **Input**: Read n (number of disks), source, target, and auxiliary.

2. **Base Case**:

* If n == 1, move disk from source to target.

3. **Recursive Steps**:

* Move n-1 disks from source to auxiliary using target as a helper.
* Move the nth disk from source to target.
* Move n-1 disks from auxiliary to target using source as a helper.

**TAHSINA SABA Roll:220622 Session:2021-22**

**Source Code:**

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

**if** n **==** 1**:**

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

**return**

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

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

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

N **=** 3

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

**Sample Input and Output:**

~Input: Enter the number of disks:

N = 3

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

~Output: Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C