

TOPIC 2:BRUTE FORCE ALGORITHM

1. List Examples

Aim: Demonstrate different types of lists.

Algorithm:

1. Create an empty list.
2. Create a list with one element.
3. Create a list with all identical elements.
4. Create a list with negative numbers and sort it.

Python code:

```
lst = eval(input("Enter a list: "))
if lst == []:
    print("Output:", lst)
else:
    print("Output:", sorted(lst))
```

Input:

[]

Output:

[]

main.py	Run	Output
<pre>1 lst = eval(input("Enter a list: ")) 2 3 if lst == []: 4 print("Output:", lst) 5 else: 6 print("Output:", sorted(lst)) 7</pre>		<pre>Enter a list: [] Output: [] === Code Execution Successful ===</pre>

2. Selection Sort

Aim: Sort an array using Selection Sort.

Algorithm:

1. Divide the array into sorted and unsorted parts.
2. Find the minimum element in the unsorted part.
3. Swap it with the first element of the unsorted part.
4. Repeat until the array is fully sorted.

Python Code:

```
def selection_sort(arr):  
    for i in range(len(arr)):  
        min_idx = i  
        for j in range(i+1, len(arr)):  
            if arr[j] < arr[min_idx]:  
                min_idx = j  
        arr[i], arr[min_idx] = arr[min_idx], arr[i]  
    return arr
```

```
arr = eval(input("Enter list to sort: "))  
print("Sorted list:", selection_sort(arr))
```

Input:

[5, 2, 9, 1, 5, 6]

Output:

[1, 2, 5, 5, 6, 9]

main.py	Run	Output
<pre>1 def selection_sort(arr): 2 for i in range(len(arr)): 3 min_idx = i 4 for j in range(i+1, len(arr)): 5 if arr[j] < arr[min_idx]: 6 min_idx = j 7 arr[i], arr[min_idx] = arr[min_idx], arr[i] 8 return arr 9 10 arr = eval(input("Enter list to sort: ")) 11 print("Sorted list:", selection_sort(arr)) 12</pre>		<pre>Enter list to sort: [5,8 ,1] Sorted list: [1, 5, 8] === Code Execution Successful ===</pre>

3. Optimized Bubble Sort

Aim: Stop Bubble Sort early if the list is already sorted.

Algorithm:

1. Compare adjacent elements and swap if needed.
2. If no swaps occur in a pass, the list is sorted.

Python Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

arr = eval(input("Enter list to sort: "))
```

```
print("Sorted list:", bubble_sort(arr))
```

Input:

[64, 25, 12, 22, 11]

[29, 10, 14, 37, 13]

[3, 5, 2, 1, 4]

[1, 2, 3, 4, 5]

[5, 4, 3, 2, 1]

Output:

[11, 12, 22, 25, 64]

[10, 13, 14, 29, 37]

[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]

main.py	Run	Output
<pre>1- def bubble_sort(arr): 2- n = len(arr) 3- for i in range(n): 4- swapped = False 5- for j in range(0, n-i-1): 6- if arr[j] > arr[j+1]: 7- arr[j], arr[j+1] = arr[j+1], arr[j] 8- swapped = True 9- if not swapped: 10- break 11- return arr 12- 13- arr = eval(input("Enter list to sort: ")) 14- print("Sorted list:", bubble_sort(arr)) 15-</pre>		<pre>Enter list to sort: [3, 8,6,2,9] Sorted list: [2, 3, 6, 8, 9] === Code Execution Successful ===</pre>

4. Insertion Sort with Duplicates

Aim: Sort arrays including duplicates.

Algorithm:

1. Take one element at a time and insert it in its correct position in the sorted part.
2. Relative order of duplicates is preserved.

Python Code:

```
def insertion_sort(arr):
```

```

for i in range(1, len(arr)):
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key:
        arr[j + 1] = arr[j]
        j -= 1
    arr[j + 1] = key
return arr

```

```

arr = eval(input("Enter list to sort: "))
print("Sorted list:", insertion_sort(arr))

```

Input:

[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

[5, 5, 5, 5, 5]

[2, 3, 1, 3, 2, 1, 1, 3]

Output:

[1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

[5, 5, 5, 5, 5]

[1, 1, 1, 2, 2, 3, 3, 3]

main.py	Output
<pre> 1 def insertion_sort(arr): 2 for i in range(1, len(arr)): 3 key = arr[i] 4 j = i - 1 5 while j >= 0 and arr[j] > key: 6 arr[j + 1] = arr[j] 7 j -= 1 8 arr[j + 1] = key 9 return arr 10 11 arr = eval(input("Enter list to sort: ")) 12 print("Sorted list:", insertion_sort(arr)) 13 </pre>	<pre> Enter list to sort: [2,9,5,3,7,1] Sorted list: [1, 2, 3, 5, 7, 9] === Code Execution Successful === </pre>

5. Kth Missing Positive

Aim: Find the kth missing positive number.

Algorithm:

1. Start from 1 and check each number.
2. Count missing numbers until k is reached.

Python Code:

```
def findKthPositive(arr, k):
```

```
    missing = []
```

```
    current = 1
```

```
    while len(missing) < k:
```

```
        if current not in arr:
```

```
            missing.append(current)
```

```
            current += 1
```

```
    return missing[-1]
```

```
arr = eval(input("Enter sorted list: "))
```

```
k = int(input("Enter k: "))
```

```
print("Kth Missing Positive Number:", findKthPositive(arr, k))
```

Input:

```
[2,3,4,7,11], k=5
```

```
[1,2,3,4], k=2
```

Output:

```
9
```

```
6
```

```
main.py  Run  Output  Clear
1 def findKthPositive(arr, k):
2     missing = []
3     current = 1
4     while len(missing) < k:
5         if current not in arr:
6             missing.append(current)
7             current += 1
8     return missing[-1]
9
10 arr = eval(input("Enter sorted list: "))
11 k = int(input("Enter k: "))
12 print("Kth Missing Positive Number:", findKthPositive(arr, k))
13
```

Enter sorted list: [4,5,1,2]
Enter k: 2
Kth Missing Positive Number: 6
=== Code Execution Successful ===

6. Peak Element ($O(\log n)$)

Aim: Find a peak element using binary search.

Algorithm:

1. Start the program.
2. Input the array of numbers.
3. Define a function `find_peak(arr, low, high, n)`:
 - Find the middle index $mid = (low + high) // 2$.
 - If `arr[mid]` is greater than or equal to both neighbors (or if it's on a boundary), then `arr[mid]` is a peak element.
 - Else if the left neighbor is greater than `arr[mid]`, recursively search the left half.
 - Otherwise, recursively search the right half.
4. Call the function with `low = 0` and `high = n - 1`.
5. Print the peak element found.
6. Stop the program.

Python Code:

```
def findPeakElement(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[mid + 1]:
            right = mid
    else:
```

```
        left = mid + 1

    return left
```

```
nums = eval(input("Enter list: "))
print("Peak element index:", findPeakElement(nums))
```

Input:

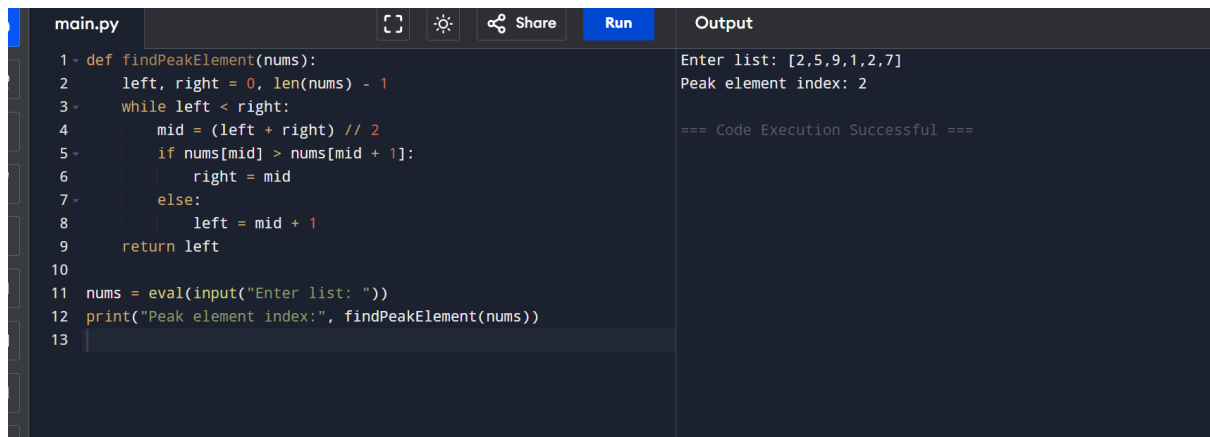
[1,2,3,1]

[1,2,1,3,5,6,4]

Output:

2

5



```
main.py  Run  Output
1 def findPeakElement(nums):
2     left, right = 0, len(nums) - 1
3     while left < right:
4         mid = (left + right) // 2
5         if nums[mid] > nums[mid + 1]:
6             right = mid
7         else:
8             left = mid + 1
9     return left
10
11 nums = eval(input("Enter list: "))
12 print("Peak element index:", findPeakElement(nums))
13
```

Enter list: [2,5,9,1,2,7]
Peak element index: 2

=== Code Execution Successful ===

7. First Occurrence of a String (Needle in Haystack)

Aim: Find the index of the first occurrence of needle in haystack.

Algorithm:

1. Loop through haystack.
2. Compare substring of length needle.
3. Return index if matched, else -1.

Python Code:

```
def strStr(haystack, needle):
    return haystack.find(needle)
```

```
haystack = input("Enter main string: ")
```



```
needle = input("Enter substring: ")
print("First occurrence index:", strStr(haystack, needle))
```


Input:

```
haystack = "sadbutsad"
```

```
needle = "sad"
```

Output:

0

A screenshot of a code editor interface. The left pane shows a file named 'main.py' with the following Python code:

```
1 - def strStr(haystack, needle):
2     return haystack.find(needle)
3
4 haystack = input("Enter main string: ")
5 needle = input("Enter substring: ")
6 print("First occurrence index:", strStr(haystack, needle))
7
```

The right pane, titled 'Output', shows the execution results:

```
Enter main string: sadbuds
Enter substring: sad
First occurrence index: 0

=== Code Execution Successful ===
```

8. Substrings in a List of Words

Aim: Return all strings that are substrings of another string in the list.

Algorithm:

1. Compare each word with all other words.
2. Add to result if it is a substring.

Python Code:

```
def stringMatching(words):
```

```
    res = []
```

```
    for i in words:
```

```
        for j in words:
```

```
            if i != j and i in j:
```

```
                res.append(i)
```

```
            break
```

```
    return res
```

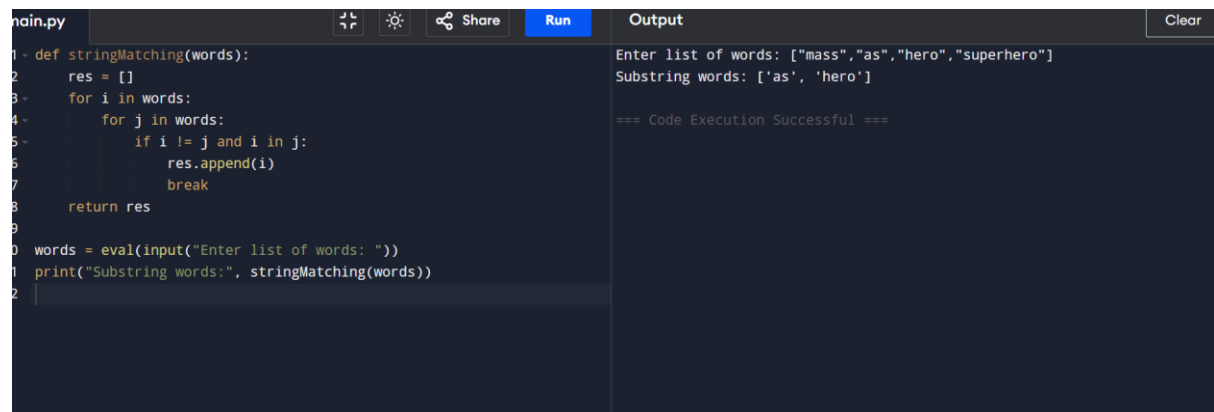
```
words = eval(input("Enter list of words: "))  
print("Substring words:", stringMatching(words))
```

Input:

```
["mass","as","hero","superhero"]
```

Output:

```
['as','hero']
```



The screenshot shows a code editor with a dark theme. On the left, the code for the `stringMatching` function is written in Python. The function iterates through each word in the input list and checks for substrings. The output panel on the right shows the input list `["mass", "as", "hero", "superhero"]` and the resulting substrings `['as', 'hero']`. A message at the bottom of the output panel states "=== Code Execution Successful ===".

```
1 def stringMatching(words):  
2     res = []  
3     for i in words:  
4         for j in words:  
5             if i != j and i in j:  
6                 res.append(i)  
7                 break  
8     return res  
9  
10 words = eval(input("Enter list of words: "))  
11 print("Substring words:", stringMatching(words))  
12
```

Enter list of words: ["mass", "as", "hero", "superhero"]
Substring words: ['as', 'hero']
=== Code Execution Successful ===

9. Closest Pair of Points (Brute Force)

Aim: Find two points with the minimum Euclidean distance.

Algorithm:

1. Check distance between all pairs.
2. Keep track of minimum distance and points.

Python Code:

```
import math  
  
def distance(p1, p2):  
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)  
  
def closest_pair(points):  
    min_dist = float('inf')  
    pair = ()  
    for i in range(len(points)):  
        for j in range(i+1, len(points)):  
            d = distance(points[i], points[j])  
            if d < min_dist:  
                min_dist = d  
                pair = (points[i], points[j])
```

```
return pair, min_dist
```

```
points = eval(input("Enter list of points: "))
```

```
pair, dist = closest_pair(points)
```

```
print("Closest pair:", pair)
```

```
print("Minimum distance:", dist)
```

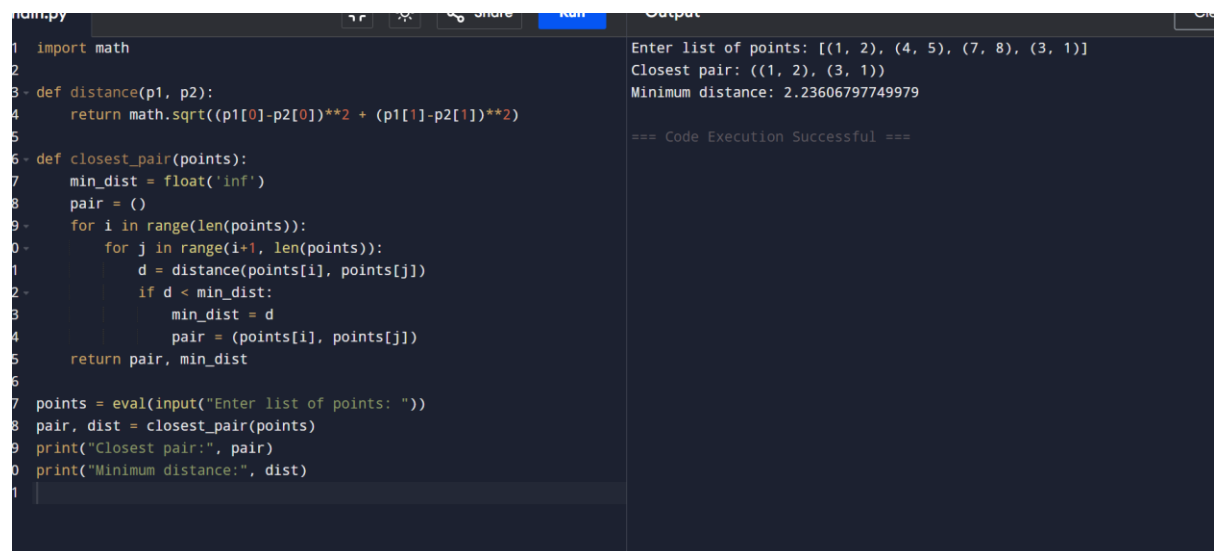
Input:

```
[(1, 2), (4, 5), (7, 8), (3, 1)]
```

Output:

```
Closest pair: ((1, 2), (3, 1))
```

```
Minimum distance: 1.4142135623730951
```



The screenshot shows a Jupyter Notebook interface with a code cell on the left and an output cell on the right. The code cell contains the following Python code:

```
1 import math
2
3 def distance(p1, p2):
4     return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)
5
6 def closest_pair(points):
7     min_dist = float('inf')
8     pair = ()
9     for i in range(len(points)):
10        for j in range(i+1, len(points)):
11            d = distance(points[i], points[j])
12            if d < min_dist:
13                min_dist = d
14                pair = (points[i], points[j])
15        return pair, min_dist
16
17 points = eval(input("Enter list of points: "))
18 pair, dist = closest_pair(points)
19 print("Closest pair:", pair)
20 print("Minimum distance:", dist)
```

The output cell displays the following text:

```
Enter list of points: [(1, 2), (4, 5), (7, 8), (3, 1)]
Closest pair: ((1, 2), (3, 1))
Minimum distance: 2.23606797749979
=== Code Execution Successful ===
```

10. Convex Hull (Brute Force)

Aim: Find convex hull of 2D points using brute force.

Algorithm:

1. A point is part of hull if all other points are on one side of the line.
2. Check all pairs of points.

Python Code:

```
def convex_hull(points):
```

```
    hull = set()
```

```

for i in range(len(points)):
    for j in range(len(points)):
        if i == j:
            continue
        a, b = points[i], points[j]
        left, right = 0, 0
        for k in range(len(points)):
            if k == i or k == j:
                continue
            x, y = points[k]
            val = (b[0]-a[0])*(y-a[1]) - (b[1]-a[1])*(x-a[0])
            if val > 0: left += 1
            elif val < 0: right += 1
        if left == 0 or right == 0:
            hull.add(a)
            hull.add(b)
    return list(hull)

```

```

points = eval(input("Enter points: "))
print("Convex Hull:", convex_hull(points))

```

Input:

```
[(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
```

Output:

```
[(0, 0), (1, 1), (8, 1), (4, 6)]
```

The screenshot shows a Python IDE with a file named 'main.py'. The code defines a function 'convex_hull(points)' that uses a brute-force method to find the convex hull of a set of points. It iterates through all pairs of points (i, j) and checks if any other point k lies to the left or right of the line segment (i, j). If so, it adds the point to the hull. The output shows the input points and the resulting convex hull.

```

1- def convex_hull(points):
2     hull = set()
3     for i in range(len(points)):
4         for j in range(len(points)):
5             if i == j:
6                 continue
7             a, b = points[i], points[j]
8             left, right = 0, 0
9             for k in range(len(points)):
10                if k == i or k == j:
11                    continue
12                x, y = points[k]
13                val = (b[0]-a[0])*(y-a[1]) - (b[1]-a[1])*(x-a[0])
14                if val > 0: left += 1
15                elif val < 0: right += 1
16                if left == 0 or right == 0:
17                    hull.add(a)
18                    hull.add(b)
19            return list(hull)
20
21 points = eval(input("Enter points: "))
22 print("Convex Hull:", convex_hull(points))
23

```

Output:

```

Enter points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
Convex Hull: [(0, 0), (4, 6), (8, 1)]

=== Code Execution Successful ===

```

11. Traveling Salesman Problem (Exhaustive Search)

Aim: Find shortest path visiting all cities.

Algorithm:

1. Generate all permutations of cities (except start).
2. Compute distance of each route.
3. Keep minimum distance and path.

Python Code:

```

import itertools, math

def distance(c1, c2):
    return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)

def tsp(cities):
    start = cities[0]
    min_dist = float('inf')
    best_path = []
    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        dist = sum(distance(path[i], path[i+1]) for i in range(len(path)-1))
        if dist < min_dist:
            min_dist = dist
            best_path = path

```

```

    return min_dist, best_path

cities = eval(input("Enter list of cities: "))

dist, path = tsp(cities)

print("Shortest Distance:", dist)

print("Shortest Path:", path)

```

Input:

[(1,2), (4,5), (7,1), (3,6)]

Output:

Shortest Distance: 7.0710678118654755

Shortest Path: [(1,2), (4,5), (7,1), (3,6), (1,2)]

The screenshot shows a code editor with a file named 'main.py'. The code defines a distance function and a tsp function that uses itertools.permutations to find the shortest path. The output window shows the input list of cities, the shortest distance, and the shortest path.

```

main.py
1 import itertools, math
2
3 def distance(c1, c2):
4     return math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)
5
6 def tsp(cities):
7     start = cities[0]
8     min_dist = float('inf')
9     best_path = []
10    for perm in itertools.permutations(cities[1:]):
11        path = [start] + list(perm) + [start]
12        dist = sum(distance(path[i], path[i+1]) for i in range(len
            (path)-1))
13        if dist < min_dist:
14            min_dist = dist
15            best_path = path
16    return min_dist, best_path
17
18 cities = eval(input("Enter list of cities: "))
19 dist, path = tsp(cities)
20 print("Shortest Distance:", dist)
21 print("Shortest Path:", path)
22
Output
Enter list of cities: [(1,2), (4,5), (7,1), (3,6)]
Shortest Distance: 16.969112047670894
Shortest Path: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]

=== Code Execution Successful ===

```

12. Assignment Problem (Exhaustive Search)

Aim: Find optimal worker-task assignment with minimum cost.

Algorithm:

1. **Start the program.**
2. **Input** the cost matrix of size $n \times n$,
where each element $\text{cost}[i][j]$ represents the cost of assigning worker i to task j .
3. **Initialize** $\text{min_cost} = \text{infinity}$ and $\text{best_assignment} = []$.
4. **Generate all possible assignments** (permutations of tasks).
5. For each assignment:
 - Calculate the **total cost** by summing $\text{cost}[i][\text{assignment}[i]]$ for all workers i .
 - If $\text{total cost} < \text{min_cost}$,

Update min_cost and store this assignment as best_assignment.

6. **Display** the optimal assignment and its total minimum cost.

7. **Stop the program.**

Python Code:

```
import itertools

def total_cost(assignment, cost_matrix):
    return sum(cost_matrix[i][assignment[i]] for i in range(len(assignment)))

def assignment_problem(cost_matrix):
    n = len(cost_matrix)
    min_cost = float('inf')
    best_assign = []
    for perm in itertools.permutations(range(n)):
        cost = total_cost(perm, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            best_assign = perm
    return best_assign, min_cost

print(assignment_problem([[3,10,7],[8,5,12],[4,6,9]]))
print(assignment_problem([[15,9,4],[8,7,18],[6,12,11]]))
```

Input:

```
[[3,10,7],[8,5,12],[4,6,9]]
[[15,9,4],[8,7,18],[6,12,11]]
```

Output:

Optimal Assignment: [1, 0, 2]

Total Cost: 19

Optimal Assignment: [2, 0, 1]

Total Cost: 24

```

main.py
14
15     for perm in permutations(range(n)):
16         cost = total_cost(perm, cost_matrix)
17         if cost < min_cost:
18             min_cost = cost
19             best_assignment = perm
20
21     print("\nOptimal Assignment:")
22     for i in range(n):
23         print(f"Worker {i+1} → Task {best_assignment[i]+1}")
24     print("Total Minimum Cost:", min_cost)
25
26 # ---- MAIN PROGRAM ----
27 n = int(input("Enter number of workers/tasks: "))
28 cost_matrix = []
29 print("Enter the cost matrix row by row:")
30
31 for i in range(n):
32     row = list(map(int, input(f"Row {i+1}: ").split()))
33     cost_matrix.append(row)
34
35 assignment_problem(cost_matrix)
36
Output
Enter number of workers/tasks: 3
Enter the cost matrix row by row:
Row 1: 2 5 7
Row 2: 4 6 8
Row 3: 2 1 1

Optimal Assignment:
Worker 1 → Task 1
Worker 2 → Task 2
Worker 3 → Task 3
Total Minimum Cost: 9

=== Code Execution Successful ===

```

13. 0-1 Knapsack Problem (Exhaustive Search)

Aim: Maximize value without exceeding capacity.

Algorithm:

1. **Start the program.**
2. **Input**
 - the list of item **weights**,
 - the list of corresponding **values**, and
 - the **maximum capacity** of the knapsack.
3. **Initialize** a variable `max_value = 0` and an empty list `best_combination = []`.
4. **For each possible subset** of the given items (use binary or combinations):
 - Calculate the **total weight** of that subset.
 - If total weight \leq capacity:
 - Compute its **total value**.
 - If total value $>$ current `max_value`,
 - Update `max_value` and store this subset as `best_combination`.
5. **Display** the indices (or names) of selected items and the total value.
6. **Stop the program.**

Python Code:

```
from itertools import combinations
```

```
def total_value(items, values):
```



```

    return sum(values[i] for i in items)
def is_feasible(items, weights, capacity):
    return sum(weights[i] for i in items) <= capacity
def knapsack(weights, values, capacity):
    n = len(weights)
    best_value = 0
    best_items = []
    for r in range(n+1):
        for items in combinations(range(n), r):
            if is_feasible(items, weights, capacity):
                val = total_value(items, values)
                if val > best_value:
                    best_value = val
                    best_items = items
    return list(best_items), best_value
print(knapsack([2,3,1],[4,5,3],4))
print(knapsack([1,2,3,4],[2,4,6,3],6))

```

Input:

Weights=[2,3,1], Values=[4,5,3], Capacity=4

Weights=[1,2,3,4], Values=[2,4,6,3], Capacity=6

Output:

Optimal Selection: [0, 2]

Total Value: 7

Optimal Selection: [0, 1, 2]

Total Value: 10

```
1 import itertools
2
3 def total_value(items, values):
4     return sum(values[i] for i in items)
5
6 def is_feasible(items, weights, capacity):
7     return sum(weights[i] for i in items) <= capacity
8
9 def knapsack(weights, values, capacity):
10     n = len(weights)
11     best_value = 0
12     best_combo = []
13     for i in range(1, n+1):
14         for combo in itertools.combinations(range(n), i):
15             if is_feasible(combo, weights, capacity):
16                 val = total_value(combo, values)
17                 if val > best_value:
18                     best_value = val
19                     best_combo = combo
20     return best_combo, best_value
21
22 weights = list(map(int, input("Enter weights: ").split()))
23 values = list(map(int, input("Enter values: ").split()))
```

```
Enter weights: 2 3 1
Enter values: 7 6 5
Enter capacity: 23
Optimal Selection: [0, 1, 2]
Total Value: 18
```

```
=== Code Execution Successful ===
```