

TOPIC 2 : BRUTE FORCE

1. Write a program to perform the following An empty list A list with one element A list with all identical elements A list with negative numbers Test Cases: 1. Input: [] • Expected Output: [] 1. Input: [1] • Expected Output: [1] 2. Input: [7, 7, 7, 7] • Expected Output: [7, 7, 7, 7] 3. Input: [-5, -1, -3, -2, -4] • Expected Output: [-5, -4, -3, -2, -1]

PROGRAM:

```
1 def f(a):
2     a.sort()
3     return a
4 print(f([]))
5 print(f([1]))
6 print(f([7,7,7,7]))
7 print(f([-5,-1,-3,-2,-4]))
```

OUTPUT:

```
[]  
[1]  
[7, 7, 7, 7]  
[-5, -4, -3, -2, -1]  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

2. Describe the Selection Sort algorithm's process of sorting an array. Selection Sort works by dividing the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements. The algorithm repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, then moves the boundary of the sorted region one element to the right. Explain why Selection Sort is simple to understand and implement but is inefficient for large datasets. Provide examples to illustrate step-by-step how Selection Sort rearranges the elements into ascending order, ensuring clarity in your explanation of the algorithm's mechanics and effectiveness. Sorting a Random Array: Input: [5, 2, 9, 1, 5, 6] Output: [1, 2, 5, 5, 6, 9] Sorting a Reverse Sorted Array: Input: [10, 8, 6, 4, 2] Output: [2, 4, 6, 8, 10] Sorting an Already Sorted Array: Input: [1, 2, 3, 4, 5] Output: [1, 2, 3, 4, 5]

PROGRAM:

```
1 def selection_sort(a):
2     n=len(a)
3     for i in range(n):
4         m=i
5     for j in range(i+1,n):
6         if a[j]<a[m]:
7             m=j
8         a[i],a[m]=a[m],a[i]
9     return a
10
11 print(selection_sort([5,2,9,1,5,6]))
12 print(selection_sort([10,8,6,4,2]))
13 print(selection_sort([1,2,3,4,5]))
14
```

OUTPUT:

```
[1, 2, 5, 5, 6, 9]
[2, 4, 6, 8, 10]
[1, 2, 3, 4, 5]

...Program finished with exit code 0
Press ENTER to exit console.
```

3. Write code to modify bubble_sort function to stop early if the list becomes sorted before all passes are completed.

PROGRAM:

```
1 def bubble_sort(a):
2     n=len(a)
3     for i in range(n):
4         s=False
5     for j in range(n-i-1):
6         if a[j]>a[j+1]:
7             a[j],a[j+1]=a[j+1],a[j]
8             s=True
9     if not s:
10        break
11    return a
12
13 print(bubble_sort([5,1,4,2,8]))
14 print(bubble_sort([1,2,3,4,5]))
```

OUTPUT:

```
[1, 2, 4, 5, 8]
[1, 2, 3, 4, 5]

...Program finished with exit code 0
Press ENTER to exit console.
```

4. Test Cases:
- Test your optimized function with the following lists:
1. Input: [64, 25, 12, 22, 11]
• Expected Output: [11, 12, 22, 25, 64]
 - 2. Input: [29, 10, 14, 37, 13] • Expected Output: [10, 13, 14, 29, 37]
 - 3. Input: [3, 5, 2, 1, 4] • Expected Output: [1, 2, 3, 4, 5]
 - 4. Input: [1, 2, 3, 4, 5] (Already sorted list) • Expected Output: [1, 2, 3, 4, 5]
 - 5. Input: [5, 4, 3, 2, 1] (Reverse sorted list) • Expected Output: [1, 2, 3, 4, 5]
- Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.
- Examples:
1. Array with Duplicates:
 - Input: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3] • Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
 2. All Identical Elements:
 - Input: [5, 5, 5, 5, 5] • Output: [5, 5, 5, 5, 5]
 - Mixed Duplicates:
 - Input: [2, 3, 1, 3, 2, 1, 1, 3] • Output: [1, 1, 1, 2, 2, 3, 3, 3]

PROGRAM:

```
1 def insertion_sort(a):
2     n=len(a)
3     for i in range(1,n):
4         key=a[i]
5         j=i-1
6         while j>=0 and a[j]>key:
7             a[j+1]=a[j]
8             j-=1
9             a[j+1]=key
10    return a
11 print(insertion_sort([64,25,12,22,11]))
12 print(insertion_sort([29,10,14,37,13]))
13 print(insertion_sort([3,5,2,1,4]))
14 print(insertion_sort([1,2,3,4,5]))
15 print(insertion_sort([5,4,3,2,1]))
16 print(insertion_sort([3,1,4,1,5,9,2,6,5,3]))
17 print(insertion_sort([5,5,5,5,5]))
18 print(insertion_sort([2,3,1,3,2,1,1,3]))
```

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

...Program finished with exit code 0
Press ENTER to exit console.
```

5. Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array. Example 1: Input: arr = [2,3,4,7,11], k = 5 Output: 9 Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9. Example 2: Input: arr = [1,2,3,4], k = 2 Output: 6 Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.

PROGRAM:

```
1 def find_kth_missing(arr,k):
2     m=0
3     num=1
4     i=0
5     while m<k:
6         if i<len(arr) and arr[i]==num:
7             i+=1
8         else:
9             m+=1
10        if m==k:
11            return num
12        num+=1
13
14 print(find_kth_missing([2,3,4,7,11],5))
15 print(find_kth_missing([1,2,3,4],2))
```

OUTPUT:

```
9
6

...Program finished with exit code 0
Press ENTER to exit console.
```

6. A peak element is an element that is strictly greater than its neighbors. Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that $\text{nums}[-1] = \text{nums}[n] = -\infty$. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in $O(\log n)$ time. Example 1: Input: nums = [1,2,3,1] Output: 2 Explanation: 3 is a peak element and your function should return the index number 2. Example 2:

Input: nums = [1,2,1,3,5,6,4] Output: 5 Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

PROGRAM:

```
1 def find_peak(nums):
2     l=0
3     r=len(nums)-1
4     while l<r:
5         m=(l+r)//2
6         if nums[m]<nums[m+1]:
7             l=m+1
8         else:
9             r=m
10    return l
11
12 print(find_peak([1,2,3,1]))
13 print(find_peak([1,2,1,3,5,6,4]))
```

OUTPUT:

```
2
5

...Program finished with exit code 0
Press ENTER to exit console.
```

7. Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack. Example 1: Input: haystack = "sadbutsad", needle = "sad" Output: 0 Explanation: "sad" occurs at index 0 and 6. The first occurrence is at index 0, so we return 0. Example 2: Input: haystack = "leetcode", needle = "leeto" Output: -1 Explanation: "leeto" did not occur in "leetcode", so we return -1.

PROGRAM:

```
1 def find_index(haystack,needle):
2     n=len(haystack)
3     m=len(needle)
4     for i in range(n-m+1):
5         if haystack[i:i+m]==needle:
6             return i
7     return -1
8
9 print(find_index("sadbutsad","sad"))
10 print(find_index("leetcode","leeto"))
```

OUTPUT:

```
0
-1

...Program finished with exit code 0
Press ENTER to exit console.
```

8. Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string Example 1: Input: words = ["mass","as","hero","superhero"] Output: ["as","hero"] Explanation: "as" is substring of "mass" and "hero" is substring of "superhero". ["hero","as"] is also a valid answer. Example 2: Input: words = ["leetcode","et","code"] Output: ["et","code"] Explanation: "et", "code" are substring of "leetcode". Example 3: Input: words = ["blue","green","bu"] Output: [] Explanation: No string of words is substring of another string.

PROGRAM:

```
1 def find_substrings(words):
2     res=[]
3     for i in range(len(words)):
4         for j in range(len(words)):
5             if i!=j and words[i] in words[j]:
6                 res.append(words[i])
7                 break
8     return res
9
10 print(find_substrings(["mass","as","hero","superhero"]))
11 print(find_substrings(["leetcode","et","code"]))
12 print(find_substrings(["blue","green","bu"]))
13
```

OUTPUT:

```
['as', 'hero']
['et', 'code']
[]

...Program finished with exit code 0
Press ENTER to exit console.
```

9. Write a program that finds the closest pair of points in a set of 2D points using the brute force approach. Input: A list or array of points represented by coordinates (x, y). Points: [(1, 2), (4, 5), (7, 8), (3, 1)] Output: The two points with the minimum distance between them. The minimum distance itself. Closest pair: (1, 2) - (3, 1) Minimum distance: 1.4142135623730951

PROGRAM:

```
1 import math
2
3 def closest_pair(points):
4     md=10**9
5     p1=p2=None
6     for i in range(len(points)):
7         for j in range(i+1, len(points)):
8             d=math.dist(points[i], points[j])
9             if d<md:
10                 md=d
11                 p1=points[i]
12                 p2=points[j]
13     return p1,p2,md
14
15 pts=[(1,2),(4,5),(7,8),(3,1)]
16 a,b,d=closest_pair(pts)
17 print("Closest pair:",a,"-",b)
18 print("Minimum distance:",d)
```

OUTPUT:

```
Closest pair: (1, 2) - (3, 1)
Minimum distance: 2.23606797749979

...Program finished with exit code 0
Press ENTER to exit console.
```

10. Write a program to find the closest pair of points in a given set using the brute force approach. Analyze the time complexity of your implementation. Define a function to calculate the Euclidean distance between two points. Implement a function to find the closest pair of points using the brute force method. Test your program with a sample set of points and verify the correctness of your results. Analyze the time complexity of your implementation. Write a brute-force algorithm to solve the convex hull problem for the following set S of points? P1 (10,0)P2 (11,5)P3 (5, 3)P4 (9, 3.5)P5 (15, 3)P6 (12.5, 7)P7 (6, 6.5)P8 (7.5, 4.5).How do you modify your brute force algorithm to handle multiple points that are lying on the sameline? Given points: P1 (10,0), P2 (11,5), P3 (5, 3), P4 (9, 3.5), P5 (15, 3), P6 (12.5, 7), P7 (6, 6.5), P8 (7.5, 4.5). output: P3, P4, P6, P5, P7, P1

PROGRAM:

```

1 import math
2 def dist(p,q):
3     return math.sqrt((p[0]-q[0])**2+(p[1]-q[1])**2)
4 def closest_pair(points):
5     md=10**9
6     a=b=None
7     for i in range(len(points)):
8         for j in range(i+1,len(points)):
9             d=dist(points[i],points[j])
10        if d<md:
11            md=d
12            a=points[i]
13            b=points[j]
14    return a,b,md
15 pts=[(1,2),(4,5),(7,8),(3,1)]
16 p1,p2,d=closest_pair(pts)
17 print("Closest pair:",p1,p2)
18 print("Minimum distance:",d)
19

```

OUTPUT:

```

Closest pair: (1, 2) (3, 1)
Minimum distance: 2.23606797749979

...Program finished with exit code 0
Press ENTER to exit console. █

```

11. Write a program that finds the convex hull of a set of 2D points using the brute force approach.
Input: A list or array of points represented by coordinates (x, y). Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)] Output: The list of points that form the convex hull in counter-clockwise order. Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

PROGRAM:

```

1 def side(a,b,c):
2     return (b[0]-a[0])*(c[1]-a[1])-(b[1]-a[1])*(c[0]-a[0])
3 def convex_hull(points):
4     hull=set()
5     n=len(points)
6     for i in range(n):
7         for j in range(n):
8             if i!=j:
9                 pos=neg=0
10                for k in range(n):
11                    if k!=i and k!=j:
12                        s=side(points[i],points[j],points[k])
13                        if s>0: pos+=1
14                        if s<0: neg+=1
15                if pos==0 or neg==0:
16                    hull.add(points[i])
17                    hull.add(points[j])
18    return sorted(hull)
19 points=[(1,1),(4,6),(8,1),(0,0),(3,3)]
20 print("Convex Hull:",convex_hull(points))
21

```

OUTPUT:

```
Convex Hull: [(0, 0), (4, 6), (8, 1)]
```

```
...Program finished with exit code 0
Press ENTER to exit console.[]
```

12. You are given a list of cities represented by their coordinates. Develop a program that utilizes exhaustive search to solve the TSP. The program should: 1. Define a function `distance(city1, city2)` to calculate the distance between two cities (e.g., Euclidean distance). 2. Implement a function `tsp(cities)` that takes a list of cities as input and performs the following:

- Generate all possible permutations of the cities (excluding the starting city) using `itertools.permutations`.
- For each permutation (representing a potential route):
 - Calculate the total distance traveled by iterating through the path and summing the distances between consecutive cities.
 - Keep track of the shortest distance encountered and the corresponding path.
- Return the minimum distance and the shortest path (including the starting city at the beginning and end).

3. Include test cases with different city configurations to demonstrate the program's functionality. Print the shortest distance and the corresponding path for each test case. Test Cases: Simple Case: Four cities with basic coordinates (e.g., `[(1, 2), (4, 5), (7, 1), (3, 6)]`) More Complex Case: Five cities with more intricate coordinates (e.g., `[(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]`) Output: Test Case 1: Shortest Distance: 7.0710678118654755 Shortest Path: `[(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]` Test Case 2: Shortest Distance: 14.142135623730951 Shortest Path: `[(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (2, 4)]`

PROGRAM:

```
1 import math
2 import itertools
3
4 def distance(a,b):
5     return math.sqrt((a[0]-b[0])**2+(a[1]-b[1])**2)
6
7 def tsp(cities):
8     start=cities[0]
9     min_dist=10**9
10    best_path=[]
11    for perm in itertools.permutations(cities[1:]):
12        path=[start]+list(perm)+[start]
13        d=0
14        for i in range(len(path)-1):
15            d+=distance(path[i],path[i+1])
16        if d<min_dist:
17            min_dist=d
18            best_path=path
19    return min_dist,best_path
20
21 cities1=[(1,2),(4,5),(7,1),(3,6)]
22 d1,p1=tsp(cities1)
23 print("Shortest Distance:",d1)
24 print("Shortest Path:",p1)
25
26 cities2=[(2,4),(8,1),(1,7),(6,3),(5,9)]
27 d2,p2=tsp(cities2)
28 print("Shortest Distance:",d2)
29 print("Shortest Path:",p2)
```

OUTPUT:

```
Shortest Distance: 16.969112047670894
Shortest Path: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]
Shortest Distance: 23.12995011084934
Shortest Path: [(2, 4), (6, 3), (8, 1), (5, 9), (1, 7), (2,
4)]

...Program finished with exit code 0
Press ENTER to exit console.
```

13. You are given a cost matrix where each element $\text{cost}[i][j]$ represents the cost of assigning worker i to task j . Develop a program that utilizes exhaustive search to solve the assignment problem. The program should Define a function $\text{total_cost}(\text{assignment}, \text{cost_matrix})$ that takes an assignment (list representing worker-task pairings) and the cost matrix as input. It iterates through the assignment and calculates the total cost by summing the corresponding costs from the cost matrix Implement a function $\text{assignment_problem}(\text{cost_matrix})$ that takes the cost matrix as input and performs the following Generate all possible permutations of worker indices (excluding repetitions). Test Cases: Input Simple Case: Cost Matrix: $[[3, 10, 7], [8, 5, 12], [4, 6, 9]]$ More Complex Case: Cost Matrix: $[[15, 9, 4], [8, 7, 18], [6, 12, 11]]$ Output: Test Case 1: Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)] Total Cost: 19 Test Case 2: Optimal Assignment: [(worker 1, task 3), (worker 2, task 1), (worker 3, task 2)] Total Cost: 24

PROGRAM:

```
1  from itertools import permutations
2
3  # Function to calculate total cost
4  def total_cost(assignment, cost_matrix):
5      total = 0
6      for i in range(len(assignment)):
7          total += cost_matrix[i][assignment[i]]
8      return total
9
10 # Function to find optimal assignment
11 def assignment_problem(cost_matrix):
12     n = len(cost_matrix)
13     all_tasks = list(range(n))
14
15     min_cost = 100000 # Large number
16     best_assign = None
17
18     for perm in permutations(all_tasks):
19         cost = total_cost(perm, cost_matrix)
20         if cost < min_cost:
21             min_cost = cost
22             best_assign = perm
23
24     # Convert to (worker, task) format starting from 1
25     result = [(i+1, best_assign[i]+1) for i in range(n)]
26     return result, min_cost
27
28 # --- Test Case 1 ---
29 cost_matrix1 = [
30     [3, 10, 7],
31     [8, 5, 12]
```

OUTPUT:

```
Test Case 1:  
Optimal Assignment: [(1, 3), (2, 2), (3, 1)]  
Total Cost: 16  
  
Test Case 2:  
Optimal Assignment: [(1, 3), (2, 2), (3, 1)]  
Total Cost: 17  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

14. You are given a list of items with their weights and values. Develop a program that utilizes exhaustive search to solve the 0-1 Knapsack Problem. The program should: Define a function total_value(items, values) that takes a list of selected items (represented by their indices) and the value list as input. It iterates through the selected items and calculates the total value by summing the corresponding values from the value list. Define a function is_feasible(items, weights, capacity) that takes a list of selected items (represented by their indices), the weight list, and the knapsack capacity as input. It checks if the total weight of the selected items exceeds the capacity. Test Cases: Simple Case: Items: 3 (represented by indices 0, 1, 2) Weights: [2, 3, 1] Values: [4, 5, 3] Capacity: 4 More Complex Case: Items: 4 (represented by indices 0, 1, 2, 3) Weights: [1, 2, 3, 4] Values: [2, 4, 6, 3] Capacity: 6 Output: Test Case 1: Optimal Selection: [0, 2] (Items with indices 0 and 2) Total Value: 7 Test Case 2: Optimal Selection: [0, 1, 2] (Items with indices 0, 1, and 2) Total Value: 10

PROGRAM:

```
1  from itertools import combinations  
2  def total_value(items, values):  
3      total=0  
4      for i in items:  
5          total+=values[i]  
6      return total  
7  def is_feasible(items, weights, capacity):  
8      total=0  
9      for i in items:  
10         total+=weights[i]  
11     return total<=capacity  
12  def knapsack(weights, values, capacity):  
13      n=len(weights)  
14      best_value=0  
15      best_items=[]  
16      for r in range(1,n+1):  
17          for combo in combinations(range(n),r):  
18              if is_feasible(combo,weights,capacity):  
19                  val=total_value(combo,values)  
20                  if val>best_value:  
21                      best_value=val  
22                      best_items=list(combo)  
23      return best_items,best_value  
24 weights1=[2,3,1]  
25 values1=[4,5,3]  
26 capacity1=4  
27 selection1,value1=knapsack(weights1,values1,capacity1)  
28 print("Test Case 1:")  
29 print("Optimal Selection:",selection1)  
30 print("Total Value:",value1)  
31 weights2=[1,2,3,4]
```

OUTPUT:

```
Test Case 1:  
Optimal Selection: [1, 2]  
Total Value: 8  
  
Test Case 2:  
Optimal Selection: [0, 1, 2]  
Total Value: 12  
  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```