

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]

AIM:

To write a C program to demonstrate different types of lists:

- Empty list
- List with one element
- List with identical elements
- List with negative numbers

ALGORITHM:

1. Start
2. Declare an integer array and variables
3. Read number of elements n
4. If n == 0, print **Empty list**

5. Else read n elements into the array
6. Display the list elements
7. Stop

PROGRAM:

```
#include <stdio.h>
```

```
int main()
{
    int a[10], n, i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    if (n == 0)
    {
        printf("Empty list");
        return 0;
    }

    printf("Enter elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("List elements are:\n");
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

INPUT:

Enter number of elements: 0

OUTPUT

Empty list

```

#include <stdio.h>
int main()
{
    int n, i, j, t;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int a[n];
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for (i = 0; i < n - 1; i++)
    {
        if (i == 0)
            t = i;
        for (j = i + 1; j < n; j++)
            if (a[j] < a[t])
                t = j;
        if (t != i)
            { t = a[i]; a[i] = a[t]; a[t] = t; }
    }
    printf("List after sorting: ");
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}

```

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.

AIM:

To write a C program to sort a given array in ascending order using the **Selection Sort** algorithm.

ALGORITHM:

1. Start
2. Read the number of elements n
3. Read n array elements
4. For each position i from 0 to n-2:
 - Assume the element at i is the smallest
 - Compare it with remaining elements
 - Find the minimum element
 - Swap it with the element at position i
5. Repeat until the array is sorted
6. Display the sorted array
7. Stop

PROGRAM:

```

#include <stdio.h>

int main()
{
    int a[10], n, i, j, t;

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    for (i = 0; i < n - 1; i++)

```

```

for (j = i + 1; j < n; j++)
    if (a[i] > a[j])
    {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

for (i = 0; i < n; i++)
    printf("%d ", a[i]);

return 0;
}

```

INPUT:

5
64 25 12 22 11

OUTPUT:

11 12 22 25 64

The screenshot shows a C IDE interface with the following details:

- Code Area:**

```

main.c
1 #include <stdio.h>
2
3 int main() {
4     int n, i, j, minIndex, temp;
5     int arr[100];
6
7     // Input
8     printf("Enter number of elements: ");
9     scanf("%d", &n);
10
11    printf("Enter the elements:\n");
12    for(i = 0; i < n; i++) {
13        scanf("%d", &arr[i]);
14    }
15
16    // Selection Sort logic
17    for(i = 0; i < n - 1; i++) {
18        minIndex = i;
19
20        for(j = i + 1; j < n; j++) {
21            if(arr[j] < arr[minIndex]) {
22                minIndex = j;
23            }
24        }
25
26        // Swap smallest element with first unsorted element
27        temp = arr[i];
28        arr[i] = arr[minIndex];
29        arr[minIndex] = temp;
30    }
31
32    // Output
33    printf("Sorted array in ascending order:\n");

```
- Output Area:**

```

Enter number of elements: 4
Enter the elements:
12 43 28 31
Sorted array in ascending order:
12 28 31 43
--- Execution Successful ---

```

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

AIM:

To write a C program to modify the Bubble Sort algorithm so that it **stops early** if the list becomes sorted before completing all passes.

ALGORITHM:

1. Start
2. Read number of elements n
3. Read n array elements
4. Repeat for each pass:
 - Set a flag swapped = 0
 - Compare adjacent elements
 - Swap if they are in wrong order and set swapped = 1
5. If no swap occurs in a pass, stop sorting
6. Display the sorted array
7. Stop

PROGRAM:

```
#include <stdio.h>
```

```
int main()
{
    int a[10], n, i, j, t, f;

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    for (i = 0; i < n - 1; i++)
    {
        f = 0;
        for (j = 0; j < n - i - 1; j++)
            if (a[j] > a[j + 1])
            {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
                f = 1;
            }
        if (f == 0) break;
    }

    for (i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

INPUT:

5
11 12 22 25 64

OUTPUT:

11 12 22 25 64

The screenshot shows a C programming environment with the following code in the main.c file:

```
main.c
1 #include <stdio.h>
2
3 void bubbleSort(int arr[], int n) {
4     int i, j, temp;
5     int swapped;
6
7     for (i = 0; i < n - 1; i++) {
8         swapped = 0; // reuse flag for next pass
9
10        for (j = 0; j < n - 1 - i; j++) {
11            if (arr[j] > arr[j + 1]) {
12                // swap
13                temp = arr[j];
14                arr[j] = arr[j + 1];
15                arr[j + 1] = temp;
16                swapped = 1;
17            }
18        }
19
20        // If no elements were swapped, array is sorted
21        if (swapped == 0)
22            break;
23    }
24 }
25
26 int main() {
27     int n, i;
28     int arr[100];
29
30     printf("Enter number of elements: ");
31     scanf("%d", &n);
32
33     printf("Enter elements(%d)\n", n);
```

The output window shows the execution results:

```
Output
> Enter number of elements: 6
Enter elements:
12
1255
12554
63
986
11
Sorted array:
11 12 43 986 1255 12554
--- Test Execution Successful ---
```

4. 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.

AIM:

To write a C program to sort an array using **Insertion Sort** and handle **duplicate elements**, preserving their relative order.

ALGORITHM:

1. Start
2. Read number of elements n
3. Read n array elements
4. For each element from index 1 to n-1:
 - Store the current element as key
 - Compare key with elements before it
 - Shift elements greater than key one position to the right
 - Insert key in its correct position
5. Display the sorted array
6. Stop

PROGRAM:

```
#include <stdio.h>

int main()
{
    int a[10], n, i, j, key;

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    for (i = 1; i < n; i++)
    {
        key = a[i];
        j = i - 1;
        while (j >= 0 && a[j] > key)
        {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }

    for (i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

INPUT:

```
6
4 3 5 3 2 4
```

OUTPUT:

2 3 3 4 4 5

The screenshot shows a C programming environment with the following code in the main.c file:

```
1 #include <stdio.h>
2
3 void insertionSort(int arr[], int n) {
4     int i, key, j;
5
6     for (i = 1; i < n; i++) {
7         key = arr[i];
8         j = i - 1;
9
10        // Move elements that are greater than key
11        // one position ahead of their current position.
12        // Elements equal to key are not moved.
13        while (j >= 0 && arr[j] > key) {
14            arr[j + 1] = arr[j];
15            j--;
16        }
17
18        // Place key at its correct position
19        arr[j + 1] = key;
20    }
21 }
22
23 int main() {
24     int n, i;
25     int arr[100];
26
27     // Input
28     printf("Enter number of elements: ");
29     scanf("%d", &n);
30
31     printf("Enter the elements: ");
32
33     for (i = 0; i < n; i++)
34         printf("%d ", arr[i]);
35 }
```

The output window shows the following interaction:

```
Enter number of elements: 4
Enter the elements:
22
5
99
49
Sorted array:
5 22 49 99
--- Done Execution Successful ---
```

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.

AIM:

To find the **kth missing positive integer** from a given array of positive integers sorted in strictly increasing order.

ALGORITHM:

1. Start
2. Read number of elements n
3. Read array elements
4. Read value of k
5. Initialize count = 0 and num = 1
6. Traverse the array:
 - If current array element ≠ num, increment count
 - If count == k, print num and stop
 - Else increment num
 - If equal, move to next array element and increment num
7. Stop

PROGRAM:

```
#include <stdio.h>
```

```
int main()
{
    int a[10], n, k, i = 0, m = 1;

    scanf("%d", &n);
    for (int j = 0; j < n; j++)
        scanf("%d", &a[j]);

    scanf("%d", &k);

    while (k)
    {
        if (i < n && a[i] == m)
```

```

        i++;
    else
        k--;
    if (k == 0)
        printf("%d", m);
    m++;
}
return 0;
}
INPUT:
5
2 3 4 7 11
5
OUTPUT:
9

```

```

min.c
1 #include <stdio.h>
2
3 int findKthMissing(int arr[], int n, int k) {
4     int current = 1;
5     int l = 0;
6
7     while (l < n) {
8         if (arr[l] == current) {
9             l++;
10        } else {
11            k--;
12            if (k == 0)
13                return current;
14        }
15        current++;
16    }
17
18    // If kth missing is beyond the last element
19    return current + k - 1;
20 }
21
22 int main() {
23     int n, k;
24     int arr[100];
25
26     printf("Enter number of elements: ");
27     scanf("%d", &n);
28
29     printf("Enter sorted array elements: ");
30     for (int i = 0; i < n; i++)
31         scanf("%d", &arr[i]);
32
33     printf("Enter k: ");

```

Output:

```

Enter number of elements: 5
Enter sorted array elements:
11
43
88
96
34
Enter k: 7
The 7th missing positive number is: 7

Code Execution Successful

```

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.

AIM:

To find a **peak element** in a given integer array and return its index using an algorithm that runs in $O(\log n)$ time.

ALGORITHM:

1. Start
2. Read number of elements n
3. Read array elements
4. Set low = 0, high = n - 1
5. While low < high:
 - Find mid = (low + high) / 2
 - If $\text{nums}[\text{mid}] < \text{nums}[\text{mid} + 1]$, move right ($\text{low} = \text{mid} + 1$)
 - Else move left ($\text{high} = \text{mid}$)
6. When loop ends, low is the peak index
7. Print the index
8. Stop

PROGRAM:

```

#include <stdio.h>

int main()
{
    int a[20], n, l = 0, h, m;

    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    h = n - 1;
    while (l < h)
    {
        m = (l + h) / 2;
        if (a[m] < a[m + 1])
            l = m + 1;
        else
            h = m;
    }

    printf("%d", l);
    return 0;
}

```

INPUT:

6
1 2 3 1 5 4

OUTPUT:

2

```

main.c
1 #include <stdio.h>
2
3 int findPeakElement(int mas[], int n) {
4     int low = 0, high = n - 1;
5
6     while (low < high) {
7         int mid = low + (high - low) / 2;
8
9         if (mas[mid] > mas[mid + 1]) {
10             // Peak is on the left side (including mid)
11             high = mid;
12         } else {
13             // Peak is on the right side
14             low = mid + 1;
15         }
16     }
17     return low; // If high and low are same
18 }
19
20 int main() {
21     int n;
22     int mas[100];
23
24     printf("Enter number of elements: ");
25     scanf("%d", &n);
26
27     printf("Enter elements: ");
28     for (int i = 0; i < n; i++)
29         scanf("%d", &mas[i]);
30
31     int peakIndex = findPeakElement(mas, n);
32
33     printf("Peak element index: %d", peakIndex);

```

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.

AIM:

To find the index of the **first occurrence of a string (needle)** in another string (haystack). If not found, return **-1**.

ALGORITHM:

1. Start
2. Read string haystack

3. Read string needle
4. For each position i in haystack:
 - Compare characters of needle with haystack starting at i
 - If all characters match, print i and stop
5. If no match is found, print -1
6. Stop

PROGRAM:

```
#include <stdio.h>

int main()
{
    char h[50], n[20];
    int i, j, k;

    scanf("%s", h);
    scanf("%s", n);

    for (i = 0; h[i]; i++)
    {
        for (j = 0, k = i; n[j] && h[k] == n[j]; j++, k++);
        if (!n[j])
        {
            printf("%d", i);
            return 0;
        }
    }
    printf("-1");
    return 0;
}
```

INPUT:

hello

abc

OUTPUT:

-1

The screenshot shows a C programming environment with the following code:

```
main:
1 #include <stdio.h>
2 #include <string.h>
3
4 int strStr(char haystack[], char needle[]) {
5     int n = strlen(haystack);
6     int m = strlen(needle);
7
8     // Edge case: empty needle
9     if (m == 0) {
10         return 0;
11     }
12
13     for (int i = 0; i <= n - m; i++) {
14         int j;
15         for (j = 0; j < m; j++) {
16             if (haystack[i + j] != needle[j]) {
17                 break;
18             }
19             if (j == m)
20                 return i; // match found
21         }
22     }
23
24 int main() {
25     char haystack[100], needle[100];
26
27     printf("Enter haystack string: ");
28     scanf("%s", haystack);
29
30     printf("Enter needle string: ");
31     scanf("%s", needle);
32
33     int index = strStr(haystack, needle);
34 }
```

The output window shows the results of running the program:

```
Output
Enter haystack string: hello
Enter needle string: ll
First occurrence index: 2
--- End Execution Successful ---
```

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

AIM:

To find and return all strings from a given array that are **substrings of another string** in the array.

ALGORITHM:

1. Start
2. Read number of strings n
3. Read n strings into array words
4. For each string words[i]:
 - Compare it with every other string words[j] where $i \neq j$
 - Check if words[i] is a substring of words[j]
 - If yes, print words[i] and stop checking further
5. Stop

PROGRAM:

```
#include <stdio.h>
#include <string.h>

int main() {
    char w[10][50];
    int n, i, j;

    // Read number of words
    scanf("%d", &n);

    // Read each word
    for (i = 0; i < n; i++)
        scanf("%s", w[i]);

    printf("Words that are substrings of other words: ");
```

```
// Compare each word with every other word
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (i != j && strstr(w[j], w[i])) {
            printf("%s ", w[i]);
            break; // Move to next word once found
        }
    }
}
```

```
printf("\n");
return 0;
}
```

INPUT:

4

mass as hero super

OUTPUT:

as hero

```

1 main()
2 #include <stdio.h>
3 #include <string.h>
4
5 int main() {
6     int n;
7     char words[100][100];
8
9     printf("Enter number of words: ");
10    scanf("%d", &n);
11
12    printf("Enter the words:\n");
13    for (int i = 0; i < n; i++) {
14        scanf("%s", words[i]);
15    }
16
17    printf("Words that are substrings of another word: ");
18
19    for (int i = 0; i < n; i++) {
20        for (int j = i + 1; j < n; j++) {
21            if (j > i) {
22                if (strcmp(words[i], words[j]) == 0) {
23                    printf("%s\n", words[i]);
24                    break; // avoid duplicate printing
25                }
26            }
27        }
28    }
29
30    return 0;
31 }

```

9. Write a program that finds the closest pair of points in a set of 2D points using the brute force approach.

AIM:

To write a C program to find the **closest pair of points** in a given set of 2D points using the **brute force method**.

ALGORITHM:

1. Start
 2. Read number of points n
 3. Read n points as (x, y) coordinates
 4. Initialize minimum distance as distance between first two points
 5. Compare every pair of points
 6. Compute distance using formula
- $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$
7. Update minimum distance if a smaller distance is found
 8. Display the closest distance
 9. Stop

PROGRAM:

```

#include <stdio.h>
#include <math.h>
int main() {
    int n, i, j;
    float x[10], y[10], d, min;

    scanf("%d", &n);
    if(n < 2) {
        printf("At least 2 points are required.\n");
        return 0;
    }
    for(i = 0; i < n; i++)
        scanf("%f %f", &x[i], &y[i]);
    min = sqrt((x[1]-x[0])*(x[1]-x[0]) + (y[1]-y[0])*(y[1]-y[0]));
    for(i = 0; i < n; i++) {
        for(j = i + 1; j < n; j++) {
            d = sqrt((x[j]-x[i])*(x[j]-x[i]) + (y[j]-y[i])*(y[j]-y[i]));

```

```

        if(d < min)
            min = d;
    }
}
printf("%.2f\n", min);
return 0;
}

```

INPUT:

```

4
1 1
2 2
4 6
7 8

```

OUTPUT:

Minimum distance = 1.41

```

1 //include <math.h>
2 //include <stdio.h>
3
4 int main() {
5     int n;
6     float x[100], y[100];
7     float minDist, dist;
8     int p1 = 0, p2 = 0;
9
10    printf("Enter number of points: ");
11    scanf("%d", &n);
12
13    printf("Enter the points (x y)\n");
14    for (int i = 0; i < n; i++) {
15        scanf("%f %f", &x[i], &y[i]);
16    }
17
18    // Initialize minimum distance
19    minDist = sqrt(pow(x[0] - x[0], 2) + pow(y[0] - y[0], 2));
20
21    // Do the暴力 comparison
22    for (int i = 0; i < n; i++) {
23        for (int j = i + 1; j < n; j++) {
24            dist = sqrt(pow(x[i] - x[j], 2) + pow(y[i] - y[j], 2));
25
26            if (dist < minDist) {
27                minDist = dist;
28                p1 = i;
29                p2 = j;
30            }
31        }
32    }
33
34    return 0;
35 }

```

Output:

```

Enter number of points: 4
Enter the points (x y):
1 2
2 6
3 2
2 1

Closest pair of points:
Point 1: (1,00, 2,00)
Point 2: (2,00, 1,00)
Minimum Distance: 1.41

*** Code Execution Successful ***

```

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

AIM:

To find the **closest pair of points** in a given set of 2D points using the **brute force approach** and analyze its time complexity.

ALGORITHM:

1. Start
2. Read number of points n
3. Read n points (x, y)
4. Define a function to compute Euclidean distance
5. Initialize minimum distance using first two points
6. Compare every pair of points
7. Update minimum distance if a smaller value is found
8. Print the minimum distance
9. Stop

PROGRAM:

```
#include <stdio.h>
#include <math.h>

// Function to calculate Euclidean distance between two points
float dist(float x1, float y1, float x2, float y2) {
    return sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));
}
```

```
}
```

```
int main() {
    int n, i, j;
    float x[10], y[10], d, min;

    printf("Enter number of points: ");
    scanf("%d", &n);

    if(n < 2) {
        printf("At least 2 points are required.\n");
        return 0;
    }

    printf("Enter coordinates (x y) of each point:\n");
    for(i = 0; i < n; i++)
        scanf("%f %f", &x[i], &y[i]);

    // Initialize min distance with first pair
    min = dist(x[0], y[0], x[1], y[1]);

    // Check all pairs
    for(i = 0; i < n; i++) {
        for(j = i + 1; j < n; j++) {
            d = dist(x[i], y[i], x[j], y[j]);
            if(d < min)
                min = d;
        }
    }
}
```

```

    printf("Minimum distance = %.2f\n", min);

    return 0;
}

```

INPUT:

4

1 1

2 2

4 6

7 8

OUTPUT:

Minimum distance = 1.41

```

main()
{
    #include <math.h>
    #include <stdio.h>
    //

    // Function to calculate Euclidean distance between two points
    // float dist(float x1, float y1, float x2, float y2) {
    //     return sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));
    // }

    int main() {
        int n, i, j;
        float x[10], y[10], d, min;
        //

        printf("Enter number of points: ");
        scanf("%d", &n);

        if(n < 2) {
            printf("At least 2 points are required.\n");
            return 0;
        }

        printf("Enter coordinates (x y) of each point:\n");
        for(i = 0; i < n; i++) {
            scanf("%f %f", &x[i], &y[i]);
        }

        // Initialization max distance with first pair
        min = dist(x[0], y[0], x[1], y[1]);

        //

        // Check all pairs
        for(i = 0; i < n; i++) {
            for(j = i + 1; j < n; j++) {
                d = dist(x[i], y[i], x[j], y[j]);
                if(d < min)
                    min = d;
            }
        }
    }
}

```

Output:

```

Enter number of points: 5
0.0
1.1
2.2
0.2
2.6
Enter coordinates (x y) of each point:
Minimum distance = 1.41
*** End Execution Successful ***

```

12. Write a program that finds the convex hull of a set of 2D points using the brute force approach.

AIM:

To write a C program to find the **convex hull** of a given set of 2D points using the **brute force method**.

ALGORITHM:

1. Start
2. Read number of points n
3. Read all points (x, y)
4. For every pair of points (Pi, Pj):
 - Check all other points lie on **one side** of the line PiPj
5. If all points are on the same side, include Pi and Pj in the hull
6. Print hull points
7. Stop

PROGRAM:

```
#include <stdio.h>

int main() {
    int n, i, j, k, pos, neg;
    float x[10], y[10];

    printf("Enter number of points: ");
    scanf("%d", &n);

    printf("Enter coordinates (x y) for each point:\n");
    for (i = 0; i < n; i++)
        scanf("%f %f", &x[i], &y[i]);

    printf("Convex Hull Edges:\n");
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            pos = neg = 0;
            for (k = 0; k < n; k++) {
                float val = (x[j] - x[i]) * (y[k] - y[i]) - (y[j] - y[i]) * (x[k] - x[i]);
                if (val > 0) pos++;
                if (val < 0) neg++;
            }
            if (pos == 0 || neg == 0)
                printf("(%.1f, %.1f) -> (%.1f, %.1f)\n", x[i], y[i], x[j], y[j]);
        }
    }
    return 0;
}
```

INPUT:

```
4
0 0
0 3
3 3
3 0
```

OUTPUT:

Convex Hull Points:

Enter coordinates (x y) for each point:

Convex Hull Edges:

```
(0.0, 0.0) -> (0.0, 2.0)
(0.0, 0.0) -> (2.0, 0.0)
(2.0, 2.0) -> (0.0, 2.0)
(2.0, 2.0) -> (2.0, 0.0)
```

```

main()
{
    #include <stdio.h>
    ...
    3 int main()
    4 {
    5     int n, i, j, k, pos, neg;
    6     float x[10], y[10];
    7
    8     printf("Enter number of points: ");
    9     scanf("%d", &n);
    10
    11     printf("Enter coordinates (x, y) for each point:\n");
    12     for (i = 0; i < n; i++)
    13         scanf("%f %f", &x[i], &y[i]);
    14
    15     printf("Convex Hull Edges:\n");
    16     for (i = 0; i < n; i++)
    17         for (j = i + 1; j < n; j++)
    18             pos = neg = 0;
    19             for (k = 0; k < n; k++)
    20                 dist_val = ((x[j] - x[k]) * (y[k] - y[i]) - (x[i] - x[k]) * (y[j] - y[k]));
    21                 if (dist_val > 0) pos++;
    22                 if (dist_val < 0) neg++;
    23             if (pos == 2) reg = 0;
    24             if (reg == 0) printf("(%.2f, %.2f) -> (%.2f, %.2f)\n", x[i], y[i], x[j], y[j]);
    25
    26     }
    27
    28     return 0;
    29 }

```

Output:

```

Enter number of points: 5
0.0
1.1
2.2
0.3
2.9
Enter coordinates (x, y) for each point:
Convex Hull Edges:
(0.0, 0.0) -> (1.0, 2.0)
(0.0, 0.0) -> (2.0, 0.0)
(1.0, 2.0) -> (0.0, 2.0)
(2.0, 2.0) -> (2.0, 0.0)

```

13. 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:
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.

AIM:

To solve the Traveling Salesman Problem (TSP) using **exhaustive search**, find the **shortest path** visiting all cities exactly once and returning to the starting city.

ALGORITHM:

1. Start
2. Read the number of cities n and their coordinates (x, y)
3. Define a function `distance(city1, city2)` to calculate Euclidean distance
4. Generate all **permutations of cities** (excluding the starting city for simplicity)
5. For each permutation, calculate the **total tour distance**
6. Keep track of the **minimum distance** and corresponding path
7. Print the **shortest distance** and **path**
8. Stop

PROGRAM:

```
#include <stdio.h>
#include <math.h>
```

```

int n, path[10], best[10];
float x[10], y[10], minDist = 1000000000;

// Function to calculate Euclidean distance between two points
float dist(int i, int j){
    return sqrt((x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])*(y[i]-y[j]));
}

// Generate all permutations of the path
void perm(int pos){
    if(pos == n){
        float d = 0;
        for(int i = 0; i < n-1; i++) d += dist(path[i], path[i+1]);
        d += dist(path[n-1], path[0]); // return to start
        if(d < minDist){
            minDist = d;
            for(int i = 0; i < n; i++) best[i] = path[i];
        }
        return;
    }
    for(int i = pos; i < n; i++){
        // Swap
        int t = path[i]; path[i] = path[pos]; path[pos] = t;
        perm(pos + 1);
        // Swap back
        t = path[i]; path[i] = path[pos]; path[pos] = t;
    }
}

```

```
int main(){
    printf("Enter number of cities: ");
    scanf("%d", &n);
    printf("Enter coordinates of each city (x y):\n");
    for(int i = 0; i < n; i++){
        scanf("%f %f", &x[i], &y[i]);
        path[i] = i;
    }

    perm(0);

    printf("Shortest distance = %.2f\nPath:\n", minDist);
    for(int i = 0; i < n; i++) printf("City%d -> ", best[i]+1);
    printf("City%d\n", best[0]+1); // Return to start
    return 0;
}
```

INPUT:

4

0 0

0 1

1 1

1 0

OUTPUT:

Shortest distance = 4.00

Path:

City1 -> City2 -> City3 -> City4 -> City1

```

1. #include <stdio.h>
2. #include <math.h>
3.
4. int n, px[10], py[10];
5. float x1[10], y1[10], mindist = 1000000000;
6.
7. // Function to calculate Euclidean distance between two points
8. float dist(int i, int j){
9.     return sqrt((x1[i]-x1[j])*(x1[i]-x1[j]) + (y1[i]-y1[j])*(y1[i]-y1[j]));
10. }
11.
12. // Generate all permutations of the path
13. void permute(int pos){
14.     if(pos == n){
15.         float d = 0;
16.         for(int i = 0; i < n-1; i++)
17.             d += dist(px[i], px[i+1]);
18.         if(d < mindist)
19.             mindist = d;
20.         for(int i = 0; i < n-1; i++)
21.             best[i] = px[i];
22.         return;
23.     }
24.     for(int i = pos; i < n; i++){
25.         // Swap
26.         int t = px[pos]; px[pos] = px[i]; px[i] = t;
27.         permute(i+1);
28.         // Swap back
29.         t = px[i]; px[i] = px[pos]; px[pos] = t;
30.     }
31. }
32.
33. int main(){
34.     printf("Enter number of cities: ");
35. }
```

14. You are given a cost matrix where each element $cost[i][j]$ 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 $total_cost(assignment, cost_matrix)$ 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 $assignment_problem(cost_matrix)$ that takes the cost matrix as input and performs the following Generate all possible permutations of worker indices (excluding repetitions).

AIM:

To solve the Assignment Problem using **exhaustive search**, finding the assignment of workers to tasks with **minimum total cost**.

ALGORITHM:

1. Start
2. Read number of workers/tasks n
3. Read the $n \times n$ cost matrix
4. Define a function $total_cost(assignment, cost_matrix)$:
 - Sum the costs for each worker-task pairing in the assignment
5. Generate all **permutations of worker indices**
6. For each permutation:
 - Calculate the total cost using $total_cost()$
 - Keep track of **minimum cost** and corresponding assignment
7. Print the **minimum total cost** and **assignment**
8. Stop

PROGRAM:

```
#include <stdio.h>
```

```
int n, cost[10][10], assign[10], best[10];
int min = 1000000000; // large initial value

// Calculate total cost of current assignment
```

```

int total() {
    int sum = 0;
    for(int i = 0; i < n; i++)
        sum += cost[i][assign[i]];
    return sum;
}

// Generate all permutations of assignments
void perm(int pos) {
    if(pos == n) {
        int t = total();
        if(t < min) {
            min = t;
            for(int i = 0; i < n; i++) best[i] = assign[i];
        }
        return;
    }
    for(int i = pos; i < n; i++) {
        // Swap positions
        int temp = assign[i]; assign[i] = assign[pos]; assign[pos] = temp;
        perm(pos + 1);
        // Swap back
        temp = assign[i]; assign[i] = assign[pos]; assign[pos] = temp;
    }
}

int main() {
    printf("Enter number of workers/tasks: ");
    scanf("%d", &n);

    printf("Enter cost matrix:\n");
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++)
            scanf("%d", &cost[i][j]);
        assign[i] = i;
    }

    perm(0);

    printf("Minimum cost = %d\nAssignment:\n");
    for(int i = 0; i < n; i++)
        printf("Worker%d -> Task%d\n", i+1, best[i]+1); // +1 for human-readable

    return 0;
}

```

INPUT:

3

9 2 7

6 4 3

5 8 1

OUTPUT:

Minimum cost = 9

Assignment:

Worker1 -> Task2

Worker2 -> Task1

Worker3 -> Task3

The screenshot shows a Java development environment with a code editor and a terminal window.

Code Editor:

```
1 //include cost.h
2
3 int n = cost[0][0]; assign[0][0] = best[0][0];
4 int min = 100000000; // large initial value
5
6 // calculate total cost of current assignment
7 int total() {
8     int sum = 0;
9     for(int i = 0; i < n; i++) {
10         sum += cost[i][assign[i]];
11     }
12     return sum;
13 }
14
15 // Generate all permutations of assignments
16 void permute_pos() {
17     if(pos == n) {
18         int t = total();
19         if(t < min) {
20             min = t;
21             for(int i = 0; i < n; i++) best[i] = assign[i];
22         }
23     }
24     for(int i = pos; i < n; i++) {
25         // Swap positions
26         int temp = assign[i]; assign[i] = assign[pos]; assign[pos] = temp;
27         permute(i+1);
28         // Swap back
29         temp = assign[i]; assign[i] = assign[pos]; assign[pos] = temp;
30     }
31 }
32
33 int main() {
34     cout << "Enter number of workers/tasks: ";
35     cin >> n;
36 }
```

Terminal Window (Output):

```
Enter number of workers/tasks: 3
Enter cost matrix:
9 2 7
6 4 3
1 8 1
#Initial cost = 9
Assignment:
worker1 -> Task2
worker2 -> Task1
worker3 -> Task3
```

15. 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.

AIM:

To solve the 0-1 Knapsack Problem using **exhaustive search**, finding the **maximum total value** of items that can fit in the knapsack without exceeding its capacity.

ALGORITHM:

1. Start
2. Read number of items n and knapsack capacity W
3. Read item weights and values
4. Generate all **subsets of items** (using binary representation)
5. For each subset:
 - Check if total weight \leq capacity (`is_feasible`)
 - If feasible, calculate total value (`total_value`)
 - Keep track of **maximum value** and corresponding item subset
6. Print **maximum value** and selected items
7. Stop

PROGRAM:

```
#include <stdio.h>
int n, W, w[10], v[10];
int total_value(int items[], int m){
    int sum = 0;
    for(int i = 0; i < m; i++)
        sum += v[items[i]];
    return sum;
}
int is_feasible(int items[], int m){
    int sum = 0;
    for(int i = 0; i < m; i++)
        sum += w[items[i]];
    return sum <= W;
}
int main(){
    printf("Enter number of items and maximum weight: ");
    scanf("%d %d", &n, &W);

    printf("Enter weights of items: ");
    for(int i = 0; i < n; i++) scanf("%d", &w[i]);

    printf("Enter values of items: ");
    for(int i = 0; i < n; i++) scanf("%d", &v[i]);

    int max_val = 0, best[10], best_count = 0;
```

```

// Exhaustive search through all subsets
for(int mask = 0; mask < (1 << n); mask++){
    int items[10], m = 0;

    for(int i = 0; i < n; i++)
        if(mask & (1 << i))
            items[m++] = i;

    if(is_feasible(items, m)){
        int val = total_value(items, m);
        if(val > max_val){
            max_val = val;
            best_count = m;
            for(int i = 0; i < m; i++) best[i] = items[i];
        }
    }
}

printf("Maximum value = %d\nItems selected: ", max_val);
for(int i = 0; i < best_count; i++)
    printf(" Item%d", best[i]+1); // +1 for human-readable item numbers
printf("\n");

return 0;
}

```

INPUT:

Enter number of items and maximum weight: 4 7

Enter weights of items: 2 3 4 5

Enter values of items: 3 4 5 6

OUTPUT:

Maximum value = 9

Items selected: Item2 Item3

The screenshot shows a C IDE interface with two panes. The left pane displays the source code for a knapsack problem solver. The right pane shows the console output where the user inputs the number of items and maximum weight, followed by the weights and values of the items, and finally the output showing the maximum value and the selected items.

```

1 //include <stdio.h>
2 int val[10], w[10], v[10];
3 int total_value(int items[], int w);
4 int sum = 0;
5 for(int i = 0; i < n; i++)
6     sum += items[i];
7 return sum;
8
9 int is_feasible(int items[], int w){
10    int sum = 0;
11    for(int i = 0; i < n; i++)
12        sum += w*items[i];
13    return sum <= w;
14}
15
16 int main(){
17    printf("Enter number of items and maximum weight: ");
18    scanf("%d %d", &n, &W);
19    printf("Enter weights of items: ");
20    for(int i = 0; i < n; i++) scanf("%d", &w[i]);
21    printf("Enter values of items: ");
22    for(int i = 0; i < n; i++) scanf("%d", &v[i]);
23    printf("Maximum value = %d\nItems selected: ", best[0].best_count);
24
25    int max_val = 0, best[10].best_count = 0;
26
27    // Exhaustive search through all subsets
28    for(int mask = 0; mask < (1 << n); mask+=1)
29        int items[10], m = 0;
30
31        for(int i = 0; i < n; i++)
32            if(mask & (1 << i))
33                items[m++] = i;
34
35        if(is_feasible(items, m))
36            int val = total_value(items, m);

```