1️⃣ Linear Search

```
#include <stdio.h>

int main() {

    int arr[] = {5, 8, 2, 10, 3};

    int n = 5, target = 10;

    int found = 0;

    for(int i = 0; i < n; i++) {

        if(arr[i] == target) {

            printf("Element %d found at index %d\n", target, i);

            found = 1;

            break;

        }

    }

    if(!found) printf("Element %d not found\n", target);

    return 0;

}
```

2️⃣ Binary Search (Array must be sorted)

```
#include <stdio.h>

int main() {

    int arr[] = {2, 3, 5, 8, 10};

    int n = 5, target = 8;

    int left = 0, right = n - 1, mid, found = 0;

    while(left <= right) {

        mid = (left + right) / 2;

        if(arr[mid] == target) {

            printf("Element %d found at index %d\n", target, mid);
```

```
            found = 1;
            break;
        } else if(arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    if(!found) printf("Element %d not found\n", target);
    return 0;
}
```

Binary Search (works for unsorted array):

```
#include <stdio.h>
int main() {
    int arr[] = {10, 2, 5, 8, 3};
    int n = 5, target = 8;
    int found = 0;
    for(int i = 0; i < n; i++) {
        if(arr[i] == target) {
            printf("Element %d found at index %d\n", target, i);
            found = 1;
            break;
        }
    }
```

```
    if(!found) printf("Element %d not found\n", target);
    return 0;
}
```

1️⃣ Insertion in an Array

```
#include <stdio.h>
int main() {
    int arr[10] = {1, 2, 4, 5}; // initial array
    int n = 4;                  // current size
    int pos = 2;                // position to insert (0-based index)
    int element = 3;            // element to insert
    if(n >= 10) {
        printf("Array is full, cannot insert.\n");
        return 0;
    }
    // Shift elements to the right
    for(int i = n; i > pos; i--) {
        arr[i] = arr[i-1];
    }
    arr[pos] = element; // insert element
    n++;                // increase size
    printf("Array after insertion: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

2️⃣ Deletion in an Array

```c
#include <stdio.h>

int main() {

   int arr[] = {1, 2, 3, 4, 5};

   int n = 5;

   int pos = 2; // position to delete (0-based index)

   if(pos >= n || pos < 0) {

      printf("Invalid position.\n");

      return 0;

   }

   // Shift elements to the left

   for(int i = pos; i < n-1; i++) {

      arr[i] = arr[i+1];

   }

   n--; // reduce size

   printf("Array after deletion: ");

   for(int i = 0; i < n; i++)

      printf("%d ", arr[i]);

   printf("\n");

   return 0;

}
```

3️⃣ Bubble Sort

```c
#include <stdio.h>

int main() {

   int arr[] = {5, 1, 4, 2, 8};

   int n = 5, temp;
```

```
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    printf("Sorted array: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

4️⃣ Bubble Sort with Flag (Optimized)

```
#include <stdio.h>
int main() {
    int arr[] = {5, 1, 4, 2, 8};
    int n = 5, temp;
    int swapped;
    for(int i = 0; i < n-1; i++) {
        swapped = 0;
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
```

```
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            swapped = 1;
          }
        }
        if(swapped == 0) break; // No swap means array is sorted
    }
    printf("Sorted array: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Merge Sort

```
#include <stdio.h>
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for(int i = 0; i < n1; i++) L[i] = arr[l + i];
    for(int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];
    int i = 0, j = 0, k = l;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
```

```
    }
    while(i < n1) arr[k++] = L[i++];
    while(j < n2) arr[k++] = R[j++];
}
void mergeSort(int arr[], int l, int r) {
    if(l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    mergeSort(arr, 0, n-1);
    printf("Sorted array: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

2️⃣ Quick Sort

```
#include <stdio.h>
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
```

```
    int i = low - 1, temp;
    for(int j = low; j < high; j++) {
        if(arr[j] <= pivot) {
            i++;
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    temp = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if(low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: ");
```

```c
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Selection Sort in C

```c
#include <stdio.h>
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    int i, j, min_idx, temp;
    // One by one move the boundary of unsorted subarray
    for(i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for(j = i+1; j < n; j++) {
            if(arr[j] < arr[min_idx])
                min_idx = j;
        }
        // Swap the found minimum element with the first element
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
    printf("Sorted array: ");
    for(i = 0; i < n; i++)
```

```c
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Heap Sort in C

```c
#include <stdio.h>
// Function to heapify a subtree rooted at index i
void heapify(int arr[], int n, int i) {
    int largest = i;       // Initialize largest as root
    int left = 2*i + 1;    // left child
    int right = 2*i + 2;   // right child
    int temp;
    // If left child is larger than root
    if(left < n && arr[left] > arr[largest])
        largest = left;
    // If right child is larger than largest
    if(right < n && arr[right] > arr[largest])
        largest = right;
    // If largest is not root
    if(largest != i) {
        temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
```

```
}
// Heap Sort function
void heapSort(int arr[], int n) {
   int temp;
   // Build max heap
   for(int i = n/2 - 1; i >= 0; i--)
      heapify(arr, n, i);
   // Extract elements from heap one by one
   for(int i = n-1; i >= 0; i--) {
      // Move current root to end
      temp = arr[0];
      arr[0] = arr[i];
      arr[i] = temp;
      // call max heapify on the reduced heap
      heapify(arr, i, 0);
   }
}
int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr)/sizeof(arr[0]);
   heapSort(arr, n);
   printf("Sorted array: ");
   for(int i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
   return 0;
```

```
}
```

5️⃣ Prim's Algorithm (Minimum Spanning Tree)

```
#include <stdio.h>

#include <limits.h>

#define V 5

int minKey(int key[], int mstSet[]) {

    int min = INT_MAX, min_index;

    for(int v = 0; v < V; v++)

        if(mstSet[v] == 0 && key[v] < min)

            min = key[v], min_index = v;

    return min_index;

}

int main() {

    int graph[V][V] = {

        {0, 2, 0, 6, 0},

        {2, 0, 3, 8, 5},

        {0, 3, 0, 0, 7},

        {6, 8, 0, 0, 9},

        {0, 5, 7, 9, 0}

    };

    int parent[V], key[V], mstSet[V] = {0};

    key[0] = 0;

    parent[0] = -1;

    for(int i = 1; i < V; i++) key[i] = INT_MAX;

    for(int count = 0; count < V-1; count++) {

        int u = minKey(key, mstSet);
```

```
        mstSet[u] = 1;
        for(int v = 0; v < V; v++)
            if(graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    printf("Edge \tWeight\n");
    for(int i = 1; i < V; i++)
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    return 0;
}
```

6️⃣ Kruskal's Algorithm (Minimum Spanning Tree)

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int u, v, w;
} Edge;
int parent[100];
int find(int i) {
    while(parent[i] != i) i = parent[i];
    return i;
}
void unionSet(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[a] = b;
```

```
}
int compare(const void *a, const void *b) {
   return ((Edge*)a)->w - ((Edge*)b)->w;
}
int main() {
   int V = 4, E = 5;
   Edge edges[] = {{0,1,10},{0,2,6},{0,3,5},{1,3,15},{2,3,4}};
   for(int i = 0; i < V; i++) parent[i] = i;
   qsort(edges, E, sizeof(Edge), compare);
   printf("Edge \tWeight\n");
   for(int i = 0; i < E; i++) {
      int u = edges[i].u, v = edges[i].v;
      if(find(u) != find(v)) {
         printf("%d - %d \t%d\n", u, v, edges[i].w);
         unionSet(u, v);
      }
   }
   return 0;
}
```

---

7 Dijkstra's Algorithm (Shortest Path)

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minDistance(int dist[], int sptSet[]) {
   int min = INT_MAX, min_index;
```

```
    for(int v = 0; v < V; v++)
        if(sptSet[v] == 0 && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}
int main() {
    int graph[V][V] = {
        {0, 10, 0, 0, 5},
        {0, 0, 1, 0, 2},
        {0, 0, 0, 4, 0},
        {7, 0, 6, 0, 0},
        {0, 3, 9, 2, 0}
    };
    int dist[V], sptSet[V] = {0};
    for(int i = 0; i < V; i++) dist[i] = INT_MAX;
    dist[0] = 0;
    for(int count = 0; count < V-1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;
        for(int v = 0; v < V; v++)
            if(!sptSet[v] && graph[u][v] && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printf("Vertex \tDistance from Source\n");
    for(int i = 0; i < V; i++) printf("%d \t%d\n", i, dist[i]);
    return 0;
```

```
}
```

8️⃣ Bellman-Ford Algorithm (Shortest Path with negative weights)

```
#include <stdio.h>
#include <limits.h>
typedef struct {
  int u, v, w;
} Edge;
int main() {
  int V = 5, E = 8;
  Edge edges[] = {{0,1,-1},{0,2,4},{1,2,3},{1,3,2},{1,4,2},{3,2,5},{3,1,1},{4,3,-3}};
  int dist[V];
  for(int i = 0; i < V; i++) dist[i] = INT_MAX;
  dist[0] = 0;
  for(int i = 1; i <= V-1; i++)
    for(int j = 0; j < E; j++)
      if(dist[edges[j].u] != INT_MAX && dist[edges[j].u] + edges[j].w < dist[edges[j].v])
        dist[edges[j].v] = dist[edges[j].u] + edges[j].w;
  // Check negative-weight cycles
  for(int j = 0; j < E; j++)
    if(dist[edges[j].u] != INT_MAX && dist[edges[j].u] + edges[j].w < dist[edges[j].v])
      printf("Graph contains negative weight cycle\n");
  printf("Vertex \tDistance from Source\n");
  for(int i = 0; i < V; i++) printf("%d \t%d\n", i, dist[i]);
  return 0;
}
```

Backtracking Example – N Queens Problem

```
#include <stdio.h>

#define N 4   // You can change this to 8 for 8-Queens

int board[N][N];

// Function to print solution

void printSolution() {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++)

            printf("%d ", board[i][j]);

        printf("\n");

    }

    printf("\n");

}

// Check if a queen can be placed on board[row][col]

int isSafe(int row, int col) {

    int i, j;

    // Check this row on left side

    for (i = 0; i < col; i++)

        if (board[row][i]) return 0;

    // Check upper diagonal on left side

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

        if (board[i][j]) return 0;

    // Check lower diagonal on left side

    for (i = row, j = col; j >= 0 && i < N; i++, j--)

        if (board[i][j]) return 0;

    return 1;
```

```
}
// Solve N Queen problem using backtracking
int solveNQUtil(int col) {
   if (col >= N) {  // All queens placed
      printSolution();
      return 1;
   }
   int res = 0;
   for (int i = 0; i < N; i++) {
      if (isSafe(i, col)) {
         board[i][col] = 1; // Place queen
         res = solveNQUtil(col + 1) || res;
         board[i][col] = 0; // Backtrack
      }
   }
   return res;
}
int main() {
   if (!solveNQUtil(0))
      printf("No solution exists\n");
   return 0;
}
```

Huffman Coding (Greedy Algorithm + Priority Queue)

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

```c
// A Huffman tree node
struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode *left, *right;
};
// A Min Heap (priority queue)
struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct MinHeapNode** array;
};
// Create a new heap node
struct MinHeapNode* newNode(char data, unsigned freq) {
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}
// Create a min heap
struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
```

```
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
MinHeapNode*));
    return minHeap;
}
// Swap two nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}
// Heapify
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;
    if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;
    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}
// Extract min node
```

```
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}
// Insert new node to min heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* node) {
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = node;
}
// Build min heap
struct MinHeap* buildMinHeap(char data[], int freq[], int size) {
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    for (int i = (minHeap->size - 2) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
    return minHeap;
```

```
}
// Build Huffman Tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;
    struct MinHeap* minHeap = buildMinHeap(data, freq, size);
    while (minHeap->size != 1) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }
    return extractMin(minHeap);
}

// Print Huffman codes
void printCodes(struct MinHeapNode* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
```

```
    if (!root->left && !root->right) {
        printf("%c: ", root->data);
        for (int i = 0; i < top; i++)
            printf("%d", arr[i]);
        printf("\n");
    }
}
// Main
int main() {
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr) / sizeof(arr[0]);
    struct MinHeapNode* root = buildHuffmanTree(arr, freq, size);
    int codes[100], top = 0;
    printf("Huffman Codes:\n");
    printCodes(root, codes, top);
    return 0;
}
```

Hashing in C – Example with Linear Probing

```
#include <stdio.h>
#define SIZE 10
int hashTable[SIZE];
// Initialize hash table
void initTable() {
    for(int i = 0; i < SIZE; i++)
        hashTable[i] = -1; // -1 indicates empty slot
```

```
}
// Hash function
int hash(int key) {
    return key % SIZE;
}
// Insert key into hash table
void insert(int key) {
    int index = hash(key);
    int originalIndex = index;
    int i = 0;
    while(hashTable[index] != -1) { // collision handling
        i++;
        index = (originalIndex + i) % SIZE;
    }
    hashTable[index] = key;
}
// Search key in hash table
int search(int key) {
    int index = hash(key);
    int originalIndex = index;
    int i = 0;
    while(hashTable[index] != -1) {
        if(hashTable[index] == key)
            return index; // found
        i++;
        index = (originalIndex + i) % SIZE;
```

```
        if(i == SIZE) break; // full loop
    }
    return -1; // not found
}
// Display hash table
void display() {
    printf("Hash Table:\n");
    for(int i = 0; i < SIZE; i++)
        printf("%d: %d\n", i, hashTable[i]);
}
int main() {
    initTable();

    insert(23);
    insert(43);
    insert(13);
    insert(27);
    display();
    int key = 13;
    int index = search(key);
    if(index != -1)
        printf("Key %d found at index %d\n", key, index);
    else
        printf("Key %d not found\n", key);
    return 0;
}
```

1️⃣ Divide and Conquer – Maximum Element

```
#include <stdio.h>

// Function to find maximum using divide and conquer

int findMax(int arr[], int low, int high) {

    // If only one element

    if(low == high)

        return arr[low];

    // If two elements, return the larger one

    if(high == low + 1)

        return (arr[low] > arr[high]) ? arr[low] : arr[high];

    // Find mid

    int mid = (low + high) / 2;

    // Recursively find max in left and right halves

    int max1 = findMax(arr, low, mid);

    int max2 = findMax(arr, mid + 1, high);

    // Return the maximum of two halves

    return (max1 > max2) ? max1 : max2;

}

int main() {

    int arr[] = {5, 2, 9, 7, 6, 3};

    int n = sizeof(arr)/sizeof(arr[0]);

    int max = findMax(arr, 0, n-1);

    printf("Maximum element is %d\n", max);


    return 0;

}
```

1️⃣ BFS – Breadth-First Search

```
#include <stdio.h>

#define MAX 5

int queue[MAX], front = -1, rear = -1;

// Queue functions

void enqueue(int x) {

   if(rear == MAX-1) return;

   if(front == -1) front = 0;

   queue[++rear] = x;

}

int dequeue() {

   if(front == -1) return -1;

   int x = queue[front++];

   if(front > rear) front = rear = -1;

   return x;

}

// BFS function

void BFS(int graph[MAX][MAX], int start) {

   int visited[MAX] = {0};

   enqueue(start);

   visited[start] = 1;

   while(front != -1) {

      int node = dequeue();

      printf("%d ", node);

      for(int i = 0; i < MAX; i++) {
```

```
            if(graph[node][i] && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}
int main() {
    int graph[MAX][MAX] = {
        {0, 1, 1, 0, 0},
        {1, 0, 1, 1, 0},
        {1, 1, 0, 1, 1},
        {0, 1, 1, 0, 1},
        {0, 0, 1, 1, 0}
    };
    printf("BFS starting from node 0: ");
    BFS(graph, 0);
    printf("\n");
    return 0;
}
```

✅ **Output (example):**

BFS starting from node 0: 0 1 2 3 4

2️⃣ DFS – Depth-First Search (using recursion)

```
#include <stdio.h>

#define MAX 5

int visited[MAX] = {0};

// DFS function

void DFS(int graph[MAX][MAX], int node) {

    visited[node] = 1;

    printf("%d ", node);

    for(int i = 0; i < MAX; i++) {

        if(graph[node][i] && !visited[i]) {

            DFS(graph, i);

        }

    }

}

int main() {

    int graph[MAX][MAX] = {

        {0, 1, 1, 0, 0},

        {1, 0, 1, 1, 0},

        {1, 1, 0, 1, 1},

        {0, 1, 1, 0, 1},

        {0, 0, 1, 1, 0}

    };

    printf("DFS starting from node 0: ");

    DFS(graph, 0);

    printf("\n");
```

```
    return 0;
}
```

✅ **Output (example):**

DFS starting from node 0: 0 1 2 3 4

1️⃣ Bruteforce Practice – Find all pairs with sum = X

```
#include <stdio.h>
int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int n = sizeof(arr)/sizeof(arr[0]);
    int X = 10;
    printf("Pairs with sum %d:\n", X);
    for(int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
            if(arr[i] + arr[j] == X)
                printf("(%d, %d)\n", arr[i], arr[j]);
        }
    }
    return 0;
}
```

2️⃣ Dynamic Programming (DP) Practice – Fibonacci Number

```
#include <stdio.h>
int main() {
    int n = 10;
    int fib[n+1];
    fib[0] = 0;
```

```
    fib[1] = 1;
    for(int i = 2; i <= n; i++)
        fib[i] = fib[i-1] + fib[i-2];
    printf("Fibonacci sequence up to %d: ", n);
    for(int i = 0; i <= n; i++)
        printf("%d ", fib[i]);
    printf("\n");
    return 0;
}
```

3️⃣ Randomized Practice – Random Array Shuffling

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    srand(time(0)); // Seed for random generator
    // Fisher-Yates shuffle
    for(int i = n-1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

```
    printf("Shuffled array: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```