**Source Code:**

#include <stdio.h>

int gcd\_iterative(int a, int b) {*// Iterative GCD*

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

// Recursive GCD

int gcd\_recursive(int a, int b) {

if (b == 0)

return a;

return gcd\_recursive(b, a % b);

}

int main() {

int a = 56, b = 98;

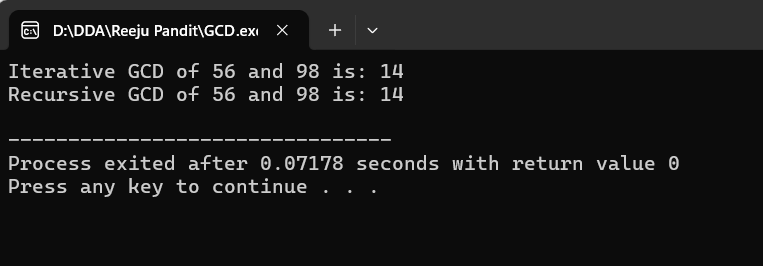
printf("Iterative GCD of %d and %d is: %d\n", a, b, gcd\_iterative(a, b));

printf("Recursive GCD of %d and %d is: %d\n", a, b, gcd\_recursive(a, b));

return 0;

}

**Output:**



**Source Code:**

#include <stdio.h>

void fibonacci\_iterative(int n) {

int a = 0, b = 1, next;

printf("Fibonacci Series up to %d terms: ", n);

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

if (i == 1) {

printf("%d ", a);

continue;

}

if (i == 2) {

printf("%d ", b);

continue;

}

next = a + b;

a = b;

b = next;

printf("%d ", next);

}

printf("\n");

}

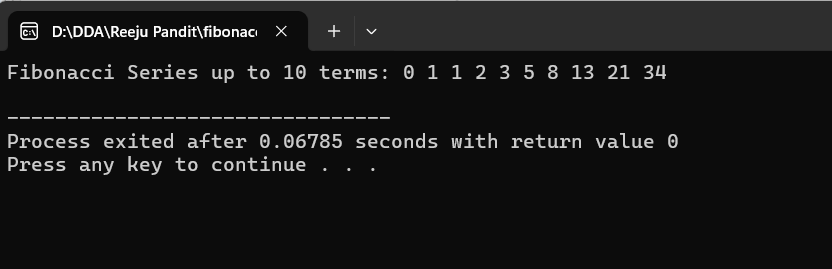
int main() {

int n = 10;

fibonacci\_iterative(n);

return 0;

}

**Output: **

**Source Code:**

#include <stdio.h>

void print\_array(int arr[], int n) {

printf("Array: ");

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

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

}

printf("\n");

}

int linear\_search(int arr[], int n, int target) {

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

if (arr[i] == target)

return i;

}

return -1;

}

int main() {

int arr[] = {5, 3, 8, 6, 2};

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

int target = 6;

int result = linear\_search(arr, n, target);

if (result != -1)

printf("Element found at index: %d\n", result);

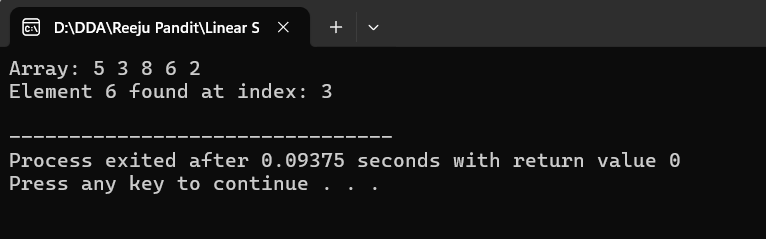
else

printf("Element not found\n");

return 0;

}

**Output:**

****

**Source code:**

#include <stdio.h>

void print\_array(int arr[], int n) {

printf("Array: ");

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

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

}

printf("\n");

}

int binary\_search\_iterative(int arr[], int n, int target) {

int left = 0, right = n - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target)

return mid;

if (arr[mid] < target)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

int binary\_search\_recursive(int arr[], int left, int right, int target) {

if (right >= left) {

int mid = left + (right - left) / 2;

if (arr[mid] == target)

return mid;

if (arr[mid] > target)

return binary\_search\_recursive(arr, left, mid - 1, target);

return binary\_search\_recursive(arr, mid + 1, right, target);

}

return -1;

}

int main() {

int arr[] = {2, 5, 7, 10, 15, 20, 25};

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

int target = 10;

print\_array(arr, n);

// Perform iterative and recursive binary searches

int iterative\_result = binary\_search\_iterative(arr, n, target);

int recursive\_result = binary\_search\_recursive(arr, 0, n - 1, target);

// Display results

if (iterative\_result != -1)

printf("Iterative Binary Search: Element %d found at index: %d\n", target, iterative\_result);

else

printf("Iterative Binary Search: Element %d not found\n", target);

if (recursive\_result != -1)

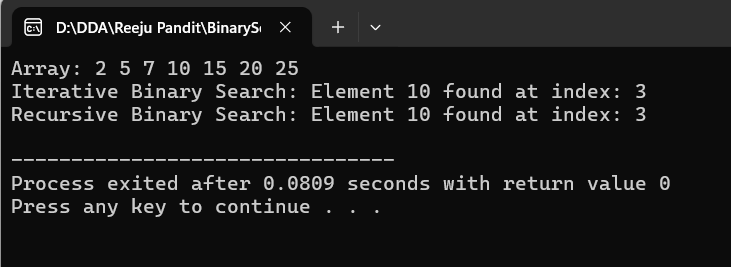
printf("Recursive Binary Search: Element %d found at index: %d\n", target, recursive\_result);

else

printf("Recursive Binary Search: Element %d not found\n", target);

return 0;

**Output:**

****

**Source Code:**

#include <stdio.h>

void print\_array(int arr[], int n) {

printf("Array: ");

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

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

}

printf("\n");

}

void selection\_sort(int arr[], int n) {

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

int min\_index = i;

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

if (arr[j] < arr[min\_index]) {

min\_index = j;

}

}

int temp = arr[min\_index];

arr[min\_index] = arr[i];

arr[i] = temp;

}

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

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

printf("Before Sorting:\n");

print\_array(arr, n);

selection\_sort(arr, n);

printf("After Sorting:\n");

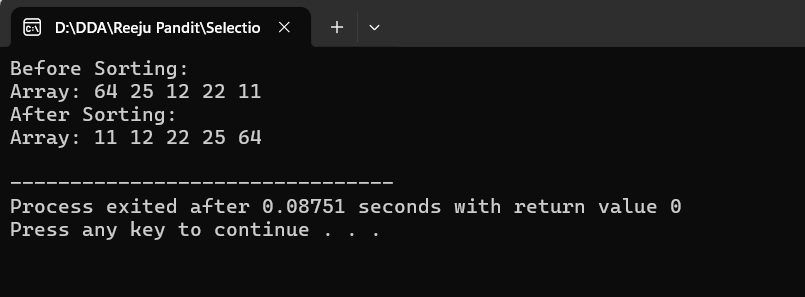
print\_array(arr, n);

printf("\n");

return 0;

}

**Output:**

****

**Source Code:**

#include <stdio.h>

void insertion\_sort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6}; // Predefined array

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

printf("Before Sorting: ");

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

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

}

printf("\n");

insertion\_sort(arr, n); // Perform the sorting

printf("After Sorting: ");

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

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

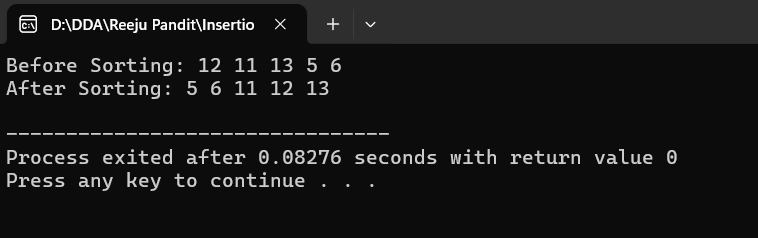
}

printf("\n");

return 0;

}

**Output:**

****

**Source Code:**

#include <stdio.h>

void min\_max\_sort(int arr[], int n) {

for (int i = 0; i < n / 2; i++) {

int min\_index = i, max\_index = i;

// Find the minimum and maximum in the remaining unsorted part

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

if (arr[j] < arr[min\_index]) {

min\_index = j;

}

if (arr[j] > arr[max\_index]) {

max\_index = j;

}

}

// Swap the minimum element with the first element

int temp = arr[i];

arr[i] = arr[min\_index];

arr[min\_index] = temp;

// Adjust the maximum index if it was swapped

if (max\_index == i) {

max\_index = min\_index;

}

// Swap the maximum element with the last element

temp = arr[n - 1 - i];

arr[n - 1 - i] = arr[max\_index];

arr[max\_index] = temp;

}

}

int main() {

int arr[] = {3, 5, 1, 10, 4}; // Predefined array

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

printf("Before Sorting: ");

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

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

}

printf("\n");

min\_max\_sort(arr, n); // Perform the sorting

printf("After Sorting: ");

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

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

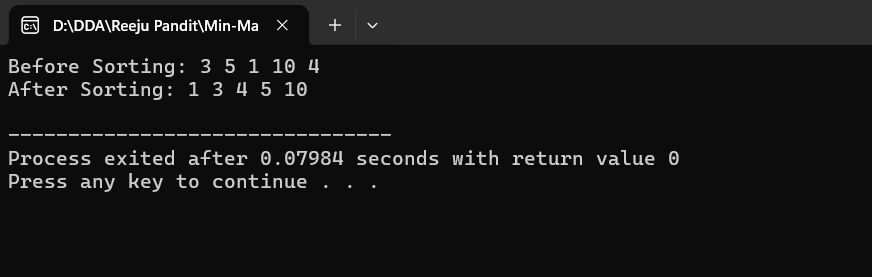
}

printf("\n");

return 0;

}

**Output:**

****

**Source code:**

#include <stdio.h>

// Merge function to combine two halves

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

// Create temporary arrays

int leftArr[n1], rightArr[n2];

// Copy data to temp arrays

for (int i = 0; i < n1; i++)

leftArr[i] = arr[left + i];

for (int i = 0; i < n2; i++)

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

// Merge the temp arrays back into the original array

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j];

j++;

}

k++;

}

// Copy remaining elements of leftArr[], if any

while (i < n1) {

arr[k] = leftArr[i];

i++;

k++;

}

// Copy remaining elements of rightArr[], if any

while (j < n2) {

arr[k] = rightArr[j];

j++;

k++;

}

}

// Merge Sort function

void merge\_sort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

merge\_sort(arr, left, mid); // Sort first half

merge\_sort(arr, mid + 1, right); // Sort second half

merge(arr, left, mid, right); // Merge both halves

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7}; // Predefined array

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

printf("Before Sorting: ");

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

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

}

printf("\n");

merge\_sort(arr, 0, n - 1); // Perform the sorting

printf("After Sorting: ");

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

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

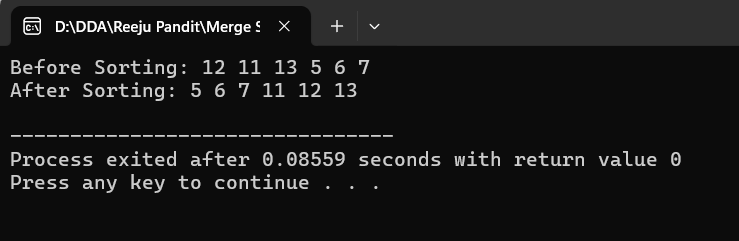
}

printf("\n");

return 0;

}

**Output:**

****

**Source Code:**

#include <stdio.h>

// Function to perform the partitioning

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

int pivot = arr[high]; // Last element as pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++; // Increment index of smaller element

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

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

arr[high] = temp;

return (i + 1);

}

// Quick Sort function

void quick\_sort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high); // Partitioning index

quick\_sort(arr, low, pi - 1); // Sort elements before partition

quick\_sort(arr, pi + 1, high); // Sort elements after partition

}

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5}; // Predefined array

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

printf("Before Sorting: ");

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

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

}

printf("\n");

quick\_sort(arr, 0, n - 1); // Perform the sorting

printf("After Sorting: ");

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

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

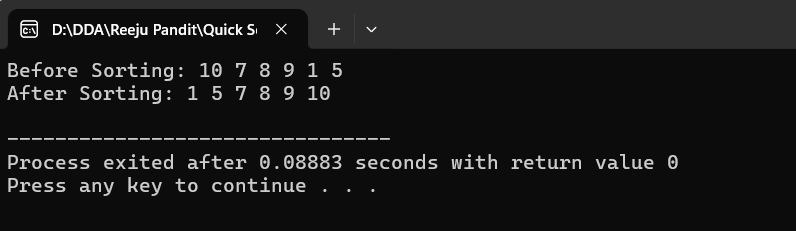
}

printf("\n");

return 0;

}

**Output:**

****

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

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

arr[high] = temp;

return (i + 1);

}

void randomized\_quick\_sort(int arr[], int low, int high) {

if (low < high) {

// Randomly select a pivot

int random = low + rand() % (high - low);

int temp = arr[random];

arr[random] = arr[high];

arr[high] = temp;

int pi = partition(arr, low, high);

randomized\_quick\_sort(arr, low, pi - 1);

randomized\_quick\_sort(arr, pi + 1, high);

}

}

void print\_array(int arr[], int n) {

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

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

}

printf("\n");

}

int main() {

srand(time(NULL)); // Seed the random number generator

int arr[] = {10, 7, 8, 9, 1, 5}; // Predefined array

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

printf("Before Sorting: ");

print\_array(arr, n); // Print the array before sorting

randomized\_quick\_sort(arr, 0, n - 1); // Perform the sorting

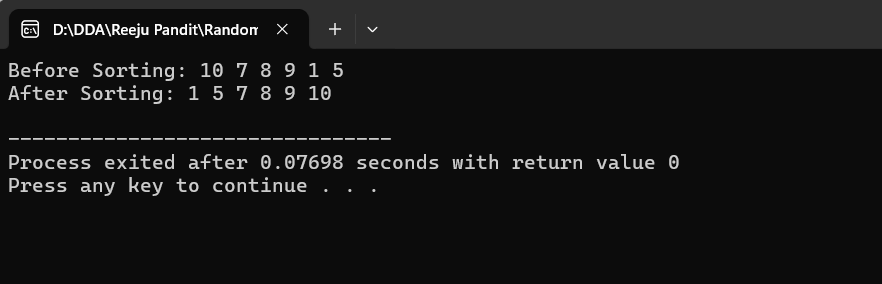
printf("After Sorting: ");

print\_array(arr, n); // Print the array after sorting

return 0;

}

**Output:**

****

**Source Code:**

#include <stdio.h>

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapify(arr, n, largest);

}

}

void heap\_sort(int arr[], int n) {

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract elements from heap

for (int i = n - 1; i >= 1; i--) {

// Move current root to end

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

void print\_array(int arr[], int n) {

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

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

}

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

printf("Before Sorting: ");

print\_array(arr, n); // Print array before sorting

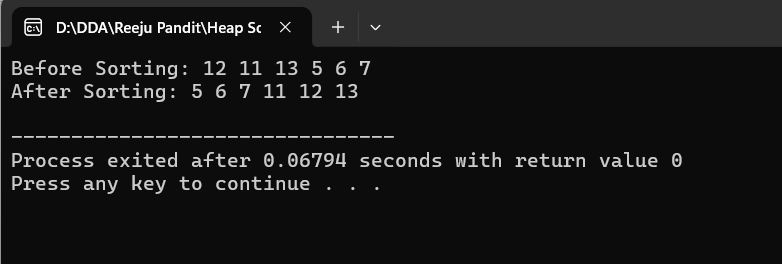
heap\_sort(arr, n); // Perform heap sort

printf("After Sorting: ");

print\_array(arr, n); // Print array after sorting

return 0;

}



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent an item

struct Item {

int value;

int weight;

};

// Comparison function to sort items based on value/weight ratio

int compare(const void\* a, const void\* b) {

double r1 = ((struct Item\*)a)->value / (double)((struct Item\*)a)->weight;

double r2 = ((struct Item\*)b)->value / (double)((struct Item\*)b)->weight;

if (r1 < r2)

return 1;

else

return -1;

}

// Function to solve the fractional knapsack problem

double fractional\_knapsack(int W, struct Item arr[], int n) {

qsort(arr, n, sizeof(arr[0]), compare); // Sort items based on value/weight ratio

int currentWeight = 0;

double finalValue = 0.0;

// Print the sorted items

printf("Sorted items by value/weight ratio:\n");

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

printf("Item %d: Value = %d, Weight = %d, Value/Weight = %.2f\n", i+1, arr[i].value, arr[i].weight, (double)arr[i].value / arr[i].weight);

}

// Iterating through items to select them for the knapsack

printf("\nItems selected:\n");

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

if (currentWeight + arr[i].weight <= W) {

currentWeight += arr[i].weight;

finalValue += arr[i].value;

printf("Full item taken: Value = %d, Weight = %d\n", arr[i].value, arr[i].weight);

} else {

int remainingWeight = W - currentWeight;

finalValue += arr[i].value \* ((double)remainingWeight / arr[i].weight);

printf("Partial item taken: Value = %.2f, Weight = %d (Fraction: %.2f)\n", arr[i].value \* ((double)remainingWeight / arr[i].weight), remainingWeight, (double)remainingWeight / arr[i].weight);

break;

}

}

return finalValue;

}

int main() {

struct Item arr[] = {{60, 10}, {100, 20}, {120, 30}};

int W = 50; // Capacity of knapsack

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

// Displaying input

printf("Input items:\n");

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

printf("Item %d: Value = %d, Weight = %d\n", i + 1, arr[i].value, arr[i].weight);

}

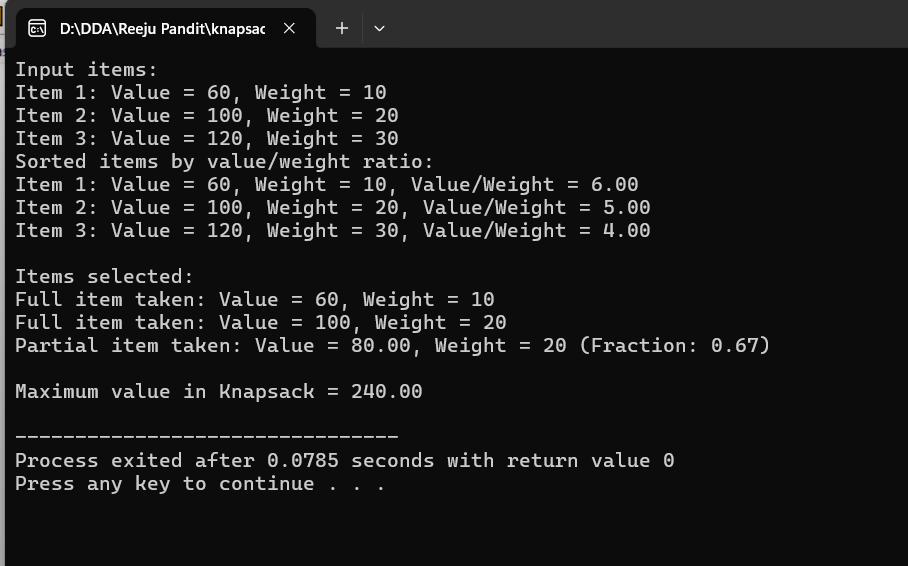
// Call the function and display result

printf("\nMaximum value in Knapsack = %.2f\n", fractional\_knapsack(W, arr, n));

return 0;

}

**Output:**

****

**Source Code:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

// Structure to represent a job

struct Job {

int id;

int deadline;

int profit;

};

// Comparison function to sort jobs in decreasing order of profit

int compare(const void\* a, const void\* b) {

return ((struct Job\*)b)->profit - ((struct Job\*)a)->profit;

}

// Function to find the job sequence with maximum profit

void job\_sequencing(struct Job arr[], int n) {

qsort(arr, n, sizeof(arr[0]), compare); // Sort jobs by profit in descending order

int result[n]; // Array to store the sequence of jobs

bool slot[n]; // Array to check if a slot is already filled

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

slot[i] = false;

// Sequence jobs in slots to maximize profit

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

// Find a free slot for this job

for (int j = arr[i].deadline - 1; j >= 0; j--) {

if (!slot[j]) {

result[j] = i;

slot[j] = true;

break;

}

}

}

// Print the job sequence with maximum profit

printf("Job sequence with maximum profit:\n");

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

if (slot[i]) {

printf("Job %d (Profit: %d) -> ", arr[result[i]].id, arr[result[i]].profit);

}

}

printf("\n");

}

int main() {

struct Job arr[] = {{1, 2, 100}, {2, 1, 19}, {3, 2, 27}, {4, 1, 25}, {5, 3, 15}};

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

// Display input jobs

printf("Input Jobs (ID, Deadline, Profit):\n");

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

printf("Job %d: Deadline = %d, Profit = %d\n", arr[i].id, arr[i].deadline, arr[i].profit);

}

// Call the job sequencing function

job\_sequencing(arr, n);

return 0;

}

**Output:**

