VIT

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

# SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

## Lab Assignment - IV, Winter Semester 2023-24

**Course Code** : BCSE204P **Slot** : L9+ L10

**Course Name:** Design and Analysis of Algorithms **Marks** : 10



## 1. Maximum Flows Algorithm
## Edmond Karp Algorithm

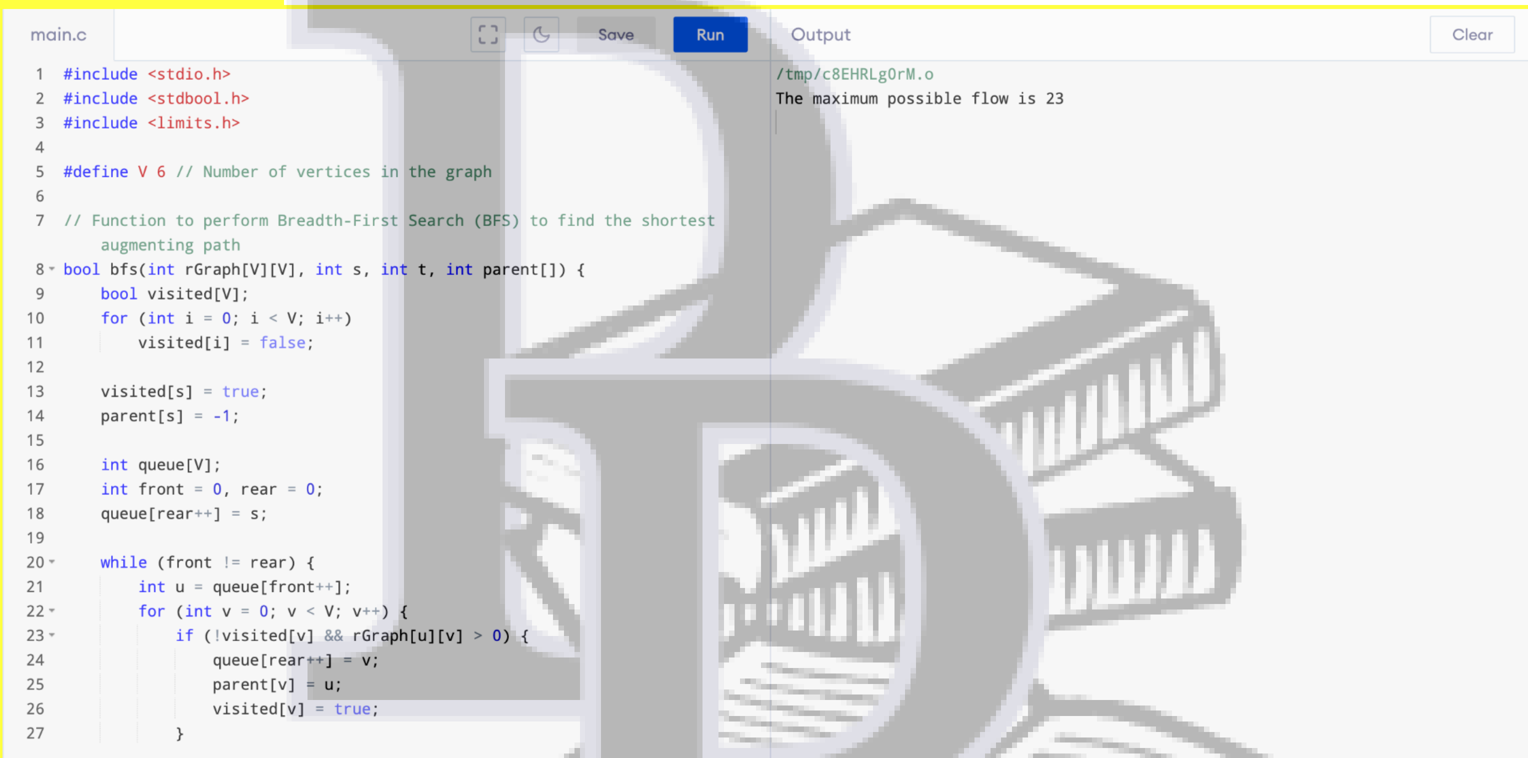
**CODE:**

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

#define V 6 // Number of vertices in the graph

// Function to perform Breadth-First Search (BFS) to find the shortest augmenting path
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
   bool visited[V];
   for (int i = 0; i < V; i++)
      visited[i] = false;

   visited[s] = true;
   parent[s] = -1;

   int queue[V];
   int front = 0, rear = 0;
   queue[rear++] = s;

   while (front != rear) {
      int u = queue[front++];
      for (int v = 0; v < V; v++) {
         if (!visited[v] && rGraph[u][v] > 0) {
```

```
            queue[rear++] = v;
            parent[v] = u;
            visited[v] = true;
         }
      }
   }
   return visited[t];
}

// Function to find the maximum flow using Edmonds-Karp algorithm
int edmondsKarp(int graph[V][V], int s, int t) {
   int u, v;

   // Residual graph where rGraph[i][j] indicates residual capacity of edge from i to j
   int rGraph[V][V];
   for (u = 0; u < V; u++)
      for (v = 0; v < V; v++)
         rGraph[u][v] = graph[u][v];

   int parent[V];
   int max_flow = 0;

   while (bfs(rGraph, s, t, parent)) {
      int path_flow = INT_MAX;
      for (v = t; v != s; v = parent[v]) {
         u = parent[v];
         path_flow = path_flow < rGraph[u][v] ? path_flow : rGraph[u][v];
      }

      for (v = t; v != s; v = parent[v]) {
         u = parent[v];
         rGraph[u][v] -= path_flow;
         rGraph[v][u] += path_flow;
      }

      max_flow += path_flow;
   }

   return max_flow;
}

// Sample usage
int main() {
   // Sample graph represented as an adjacency matrix
   int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                  {0, 0, 10, 12, 0, 0},
                  {0, 4, 0, 0, 14, 0},
                  {0, 0, 9, 0, 0, 20},
                  {0, 0, 0, 7, 0, 4},
                  {0, 0, 0, 0, 0, 0}
```

```
        };

    int source = 0, sink = 5;
    printf("The maximum possible flow is %d\n", edmondsKarp(graph, source, sink));

    return 0;
}
```

**CODE SCREENSHOT:**



```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define V 6 // Number of vertices in the graph

// Function to perform Breadth-First Search (BFS) to find the shortest
    augmenting path
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    visited[s] = true;
    parent[s] = -1;

    int queue[V];
    int front = 0, rear = 0;
    queue[rear++] = s;

    while (front != rear) {
        int u = queue[front++];
        for (int v = 0; v < V; v++) {
            if (!visited[v] && rGraph[u][v] > 0) {
                queue[rear++] = v;
                parent[v] = u;
                visited[v] = true;
            }
```



**OUTPUT:**

The maximum possible flow is 23

**OUTPUT SCREENSHOT:**





## 2. Convex Hull Finding Algorithms

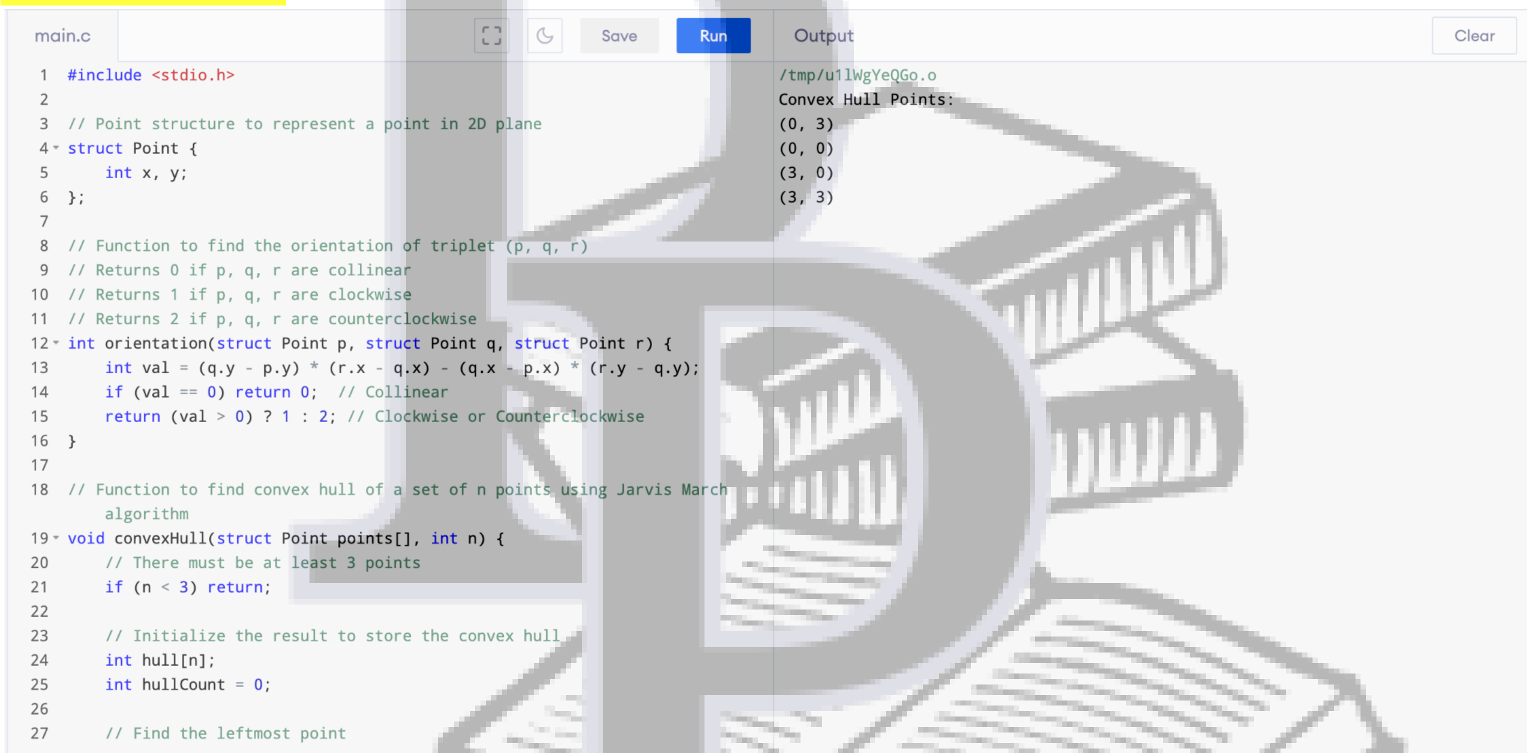
### i) Jarvis March

**CODE:**

```
#include <stdio.h>

// Point structure to represent a point in 2D plane
struct Point {
    int x, y;
};

// Function to find the orientation of triplet (p, q, r)
// Returns 0 if p, q, r are collinear
// Returns 1 if p, q, r are clockwise
// Returns 2 if p, q, r are counterclockwise
int orientation(struct Point p, struct Point q, struct Point r) {
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0) return 0;  // Collinear
    return (val > 0) ? 1 : 2; // Clockwise or Counterclockwise
}

// Function to find convex hull of a set of n points using Jarvis March algorithm
void convexHull(struct Point points[], int n) {
    // There must be at least 3 points
    if (n < 3) return;

    // Initialize the result to store the convex hull
    int hull[n];
    int hullCount = 0;

    // Find the leftmost point
    int l = 0;
    for (int i = 1; i < n; i++) {
        if (points[i].x < points[l].x)
            l = i;
    }

    // Start from the leftmost point, keep moving clockwise until we reach the start point again
    int p = l, q;
    do {
        hull[hullCount++] = p; // Add current point to convex hull
        q = (p + 1) % n;
        for (int i = 0; i < n; i++) {
            // If point q is more counterclockwise than i, update q
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;
        }
        p = q;
    } while (p != l);

    // Print the convex hull
    printf("Convex Hull Points:\n");
```

```
    for (int i = 0; i < hullCount; i++)
        printf("(%d, %d)\n", points[hull[i]].x, points[hull[i]].y);
}

// Sample usage
int main() {
    struct Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1}, {3, 0}, {0, 0}, {3, 3}};
    int n = sizeof(points) / sizeof(points[0]);
    convexHull(points, n);
    return 0;
}
```

**CODE SCREENSHOT:**




**OUTPUT:**

Convex Hull Points:
(0, 3)
(0, 0)
(3, 0)
(3, 3)



**OUTPUT SCREENSHOT:**

Output Clear

```
/tmp/u1lWgYeQGo.o
Convex Hull Points:
(0, 3)
(0, 0)
(3, 0)
(3, 3)
```

**ii) Graham's Scan**

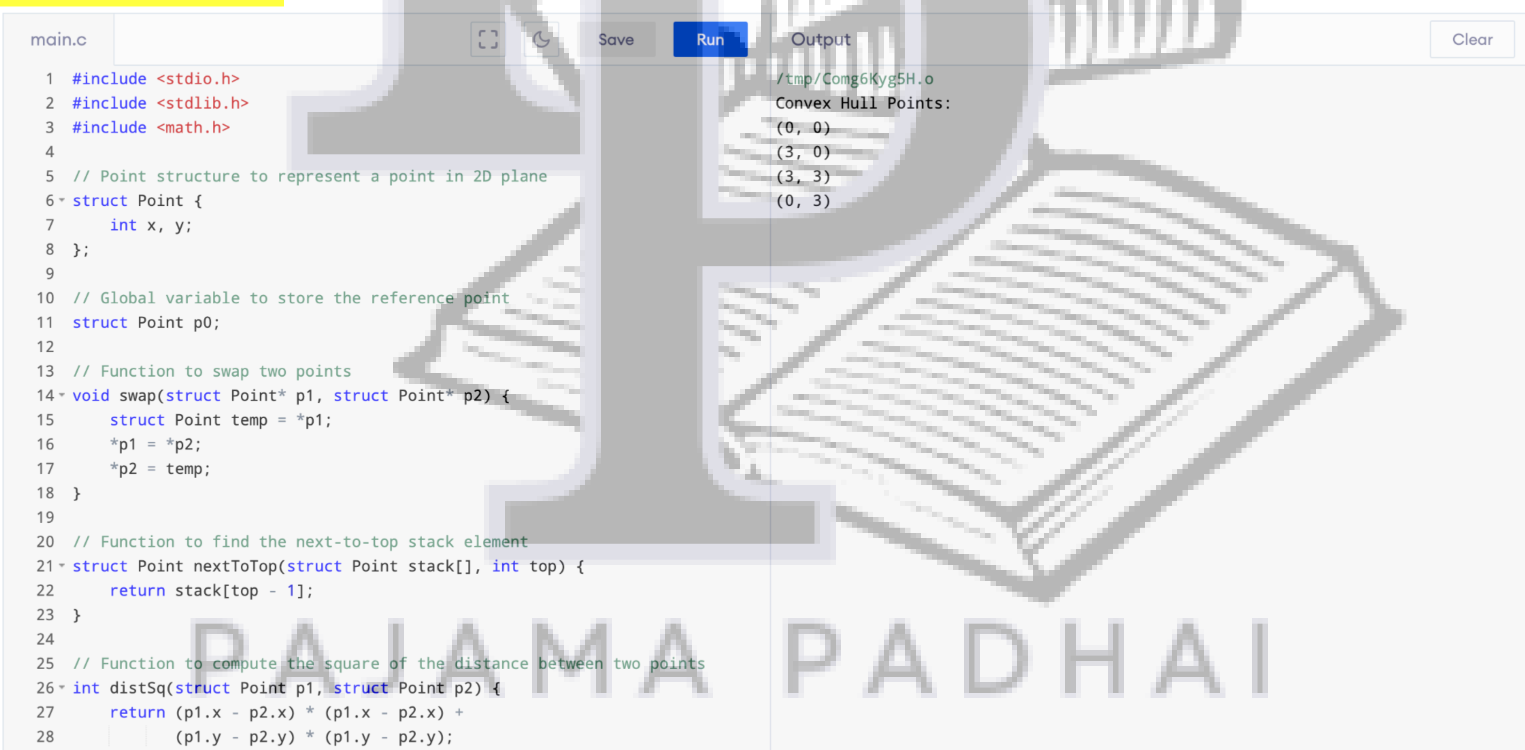
**CODE:**

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

// Point structure to represent a point in 2D plane
struct Point {
   int x, y;
};

// Global variable to store the reference point
struct Point p0;

// Function to swap two points
void swap(struct Point* p1, struct Point* p2) {
   struct Point temp = *p1;
   *p1 = *p2;
   *p2 = temp;
}

// Function to find the next-to-top stack element
struct Point nextToTop(struct Point stack[], int top) {
   return stack[top - 1];
}

// Function to compute the square of the distance between two points
int distSq(struct Point p1, struct Point p2) {
   return (p1.x - p2.x) * (p1.x - p2.x) +
         (p1.y - p2.y) * (p1.y - p2.y);
}

// Function to find the orientation of triplet (p, q, r)
// Returns 0 if p, q, r are collinear
// Returns 1 if p, q, r are clockwise
```

```
// Returns 2 if p, q, r are counterclockwise
int orientation(struct Point p, struct Point q, struct Point r) {
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0) return 0;  // Collinear
    return (val > 0) ? 1 : 2; // Clockwise or Counterclockwise
}

// Function to compare two points based on polar angle
int compare(const void* vp1, const void* vp2) {
    struct Point* p1 = (struct Point*)vp1;
    struct Point* p2 = (struct Point*)vp2;

    int orient = orientation(p0, *p1, *p2); // Renaming the variable to avoid conflict
    if (orient == 0)
        return (distSq(p0, *p2) >= distSq(p0, *p1)) ? -1 : 1;

    return (orient == 2) ? -1 : 1;
}

// Function to perform Graham's Scan algorithm to find convex hull
void grahamScan(struct Point points[], int n) {
    // Find the point with the lowest y-coordinate (and the leftmost one in case of ties)
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++) {
        int y = points[i].y;

        // Pick the bottom-most or choose the left most point in case of tie
        if ((y < ymin) || (ymin == y && points[i].x < points[min].x))
            ymin = points[i].y, min = i;
    }

    // Place the bottom-most point at first position
    swap(&points[0], &points[min]);

    // Initialize reference point
    p0 = points[0];

    // Sort the remaining points based on polar angle
    qsort(&points[1], n - 1, sizeof(struct Point), compare);

    // Initialize stack
    struct Point stack[n];
    int top = 3;
    stack[0] = points[0];
    stack[1] = points[1];
    stack[2] = points[2];

    // Process remaining n-3 points
    for (int i = 3; i < n; i++) {
```

```
    // Keep removing top while the angle formed by points stack[top], points[i], and stack[top-1] is not counterclockwise
    while (orientation(nextToTop(stack, top), stack[top], points[i]) != 2)
      top--;

    // Add point[i] to the stack
    stack[++top] = points[i];
  }

  // Print points in the convex hull
  printf("Convex Hull Points:\n");
  for (int i = 0; i <= top; i++)
    printf("(%d, %d)\n", stack[i].x, stack[i].y);
}

// Sample usage
int main() {
  struct Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1}, {3, 0}, {0, 0}, {3, 3}};
  int n = sizeof(points) / sizeof(points[0]);
  grahamScan(points, n);
  return 0;
}
```

**CODE SCREENSHOT:**

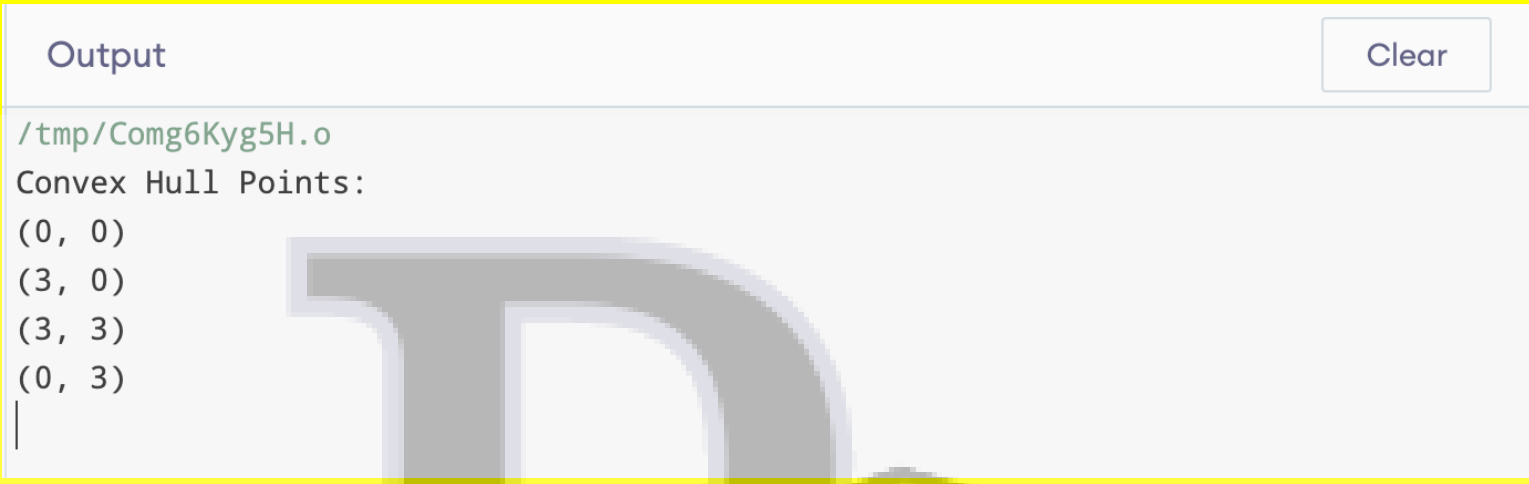



**OUTPUT:**

Convex Hull Points:
(0, 0)
(3, 0)
(3, 3)

(0, 3)

**OUTPUT SCREENSHOT:**




### 3. Randomized Algorithms
### Randomized Quick Sort

**CODE:**

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

// Function to swap two elements in an array
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

// Function to partition the array using a random pivot
int partition(int arr[], int low, int high) {
   // Selecting a random pivot
   srand(time(NULL));
   int randomIndex = low + rand() % (high - low + 1);
   int pivot = arr[randomIndex];

   // Place the pivot element at its correct position
   swap(&arr[randomIndex], &arr[high]);

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

   for (int j = low; j < high; j++) {
      // If current element is smaller than or equal to the pivot
      if (arr[j] <= pivot) {
         i++; // Increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
```

```
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to perform Quick Sort recursively
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Sample usage
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

**CODE SCREENSHOT:**

main.c

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

// Function to swap two elements in an array
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to partition the array using a random pivot
int partition(int arr[], int low, int high) {
    // Selecting a random pivot
    srand(time(NULL));
    int randomIndex = low + rand() % (high - low + 1);
    int pivot = arr[randomIndex];

    // Place the pivot element at its correct position
    swap(&arr[randomIndex], &arr[high]);

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

    for (int j = low; j < high; j++) {
        // If current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]);
```

Output

```
/tmp/YqYxFVWgy0.o
Original array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
```

**OUTPUT:**

Original array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10

**OUTPUT SCREENSHOT:**

Output

```
/tmp/YqYxFVWgy0.o
Original array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
```