**Program-1: WAP to implement Insertion sort**

**Theory:**

Insertion sort is a simple sorting algorithm that works by iteratively building a sorted subarray starting from the leftmost element. It compares each element with the elements to its left and inserts it in the correct position to maintain the sorted order.

**Complexity:**

**Best case:** O(n) (when the array is already sorted)

**Average case:** O(n^2)

**Worst case:** O(n^2) (when the array is sorted in reverse order)

**Algorithm**:

**Initialize:** Start with the second element (index 1).

**Compare and insert:**

* Compare the current element with the elements to its left.
* If the current element is smaller than the previous element, shift the previous element to the right.
* Repeat this process until the correct position for the current element is found.
* Insert the current element in the found position.

**Iterate:** Move to the next element and repeat steps 2 and 3 until the entire array is sorted.

**Code:**

#include <stdio.h>

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

int i, key, j;

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

key = arr[i];

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};

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

insertion\_sort(arr, n);

printf("Sorted array: \n");

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

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

printf("\n");

return 0;

}

**Output:**

****

**Program-2: WAP to implement Binary Search**

**Theory:**

Binary search is a searching algorithm that efficiently locates a target element within a sorted array. It works by repeatedly dividing the search interval in half until the target element is found or determined to be absent.

**Complexity:**

**Best case**: O(1) (when the target element is the middle element)

**Average case**: O(log n)

**Worst case:** O(log n

**Algorithm:**

**Initialize:** Set left to 0 and right to the array size minus 1.

**Check middle element:** Calculate the middle index mid as (left + right) / 2.

**Compare:**

* If the target element is at mid, return mid.
* If the target element is less than arr[mid], update right to mid - 1.
* If the target element is greater than arr[mid], update left to mid + 1.

**Repeat:** Repeat steps 2 and 3 until left becomes greater than right. If the loop terminates without finding the target, it means the target is not present in the array.

**Code:**

#include <stdio.h>

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

int left = 0;

int right = n - 1;

while (left <= right) {

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

if (arr[mid] == x)

return mid;

if (arr[mid] < x)

left = mid + 1;

else

right = mid - 1;

}

return -1; // Target not found

}

int main() {

int arr[] = {2, 3, 4, 10, 40};

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

int x = 10;

int result = binary\_search(arr, n, x);

if (result == -1)

printf("Element is not present in array\n");

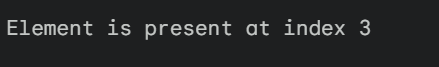
else

printf("Element is present at index %d\n", result);

return 0;

}

**Output:**

****

**Program-3: WAP to implement Selection Sort**

**Theory:**

**Selection Sort** is a simple comparison-based sorting algorithm that divides the array into two parts: a sorted part and an unsorted part. Initially, the sorted part is empty, and the unsorted part is the entire array. The algorithm repeatedly selects the smallest (or largest, depending on the sorting order) element from the unsorted part and swaps it with the leftmost unsorted element, moving the boundary between sorted and unsorted sections

**Time complexity:**

**Best-case complexity: O(n²**)  
This happens when the array is already sorted, but the algorithm still checks every element in each pass.

**Worst-case complexity: O(n²)**  
This occurs when the array is sorted in reverse order because every comparison still needs to be made.

**Average-case complexity: O(n²)**  
This is because, regardless of the input, the algorithm always performs O(n²) comparisons.

**Space complexity: O(1)**  
Selection Sort is an in-place algorithm, meaning it requires a constant amount of additional memory.

**Stability: Not stable**  
Selection Sort is not a stable sorting algorithm because it may change the relative order of elements with equal values during swaps.

**Algorithm:**

1. Start with the first element, assuming it is the smallest.
2. Compare this element with all the other elements in the array.
3. If a smaller element is found, set it as the new minimum.
4. After one full pass through the array, swap the minimum element found with the first element of the unsorted part.
5. Move to the next element and repeat the process for the remaining unsorted part of the array.
6. Continue this process until the entire array is sorted.

**Code**:

#include <stdio.h>

// Function to swap two elements

void swap(int \*xp, int \*yp) {

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

// Function to perform selection sort

void selectionSort(int arr[], int n) {

int i, j, min\_index;

// One by one move the boundary of the unsorted subarray

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

// Find the minimum element in the unsorted array

min\_index = i;

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

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

min\_index = j;

}

}

// Swap the found minimum element with the first element

swap(&arr[min\_index], &arr[i]);

}

}

// Function to print the array

void printArray(int arr[], int size) {

int i;

for (i = 0; i < size; i++) {

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

}

printf("\n");

}

int main() {

int arr[] = {65, 55, 88, 54, 50, 68, 62, 48, 31, 21, 52};

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

printf("Original array: \n");

printArray(arr, n);

// Perform selection sort

selectionSort(arr, n);

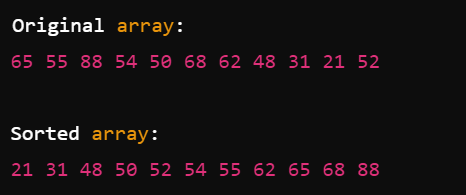
printf("\nSorted array: \n");

printArray(arr, n);

return 0;

}

**Output**:



**Program-4: WAP to implement Merge Sort**

**Theory**:

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts both halves, and then merges the sorted halves back together. The merging process involves comparing the smallest unmerged elements from each half and placing the smaller one into the new sorted array.

Characteristics:

* Stable: Maintains the relative order of equal elements.
* Time Complexity: O(n log n) for all cases (best, average, and worst).
* Space Complexity: O(n) due to the temporary arrays used during merging.

**Complexity:**

* Time: O(n log n) for all cases (best, average, worst)
* Space: O(n) (for the temporary array)

**Algorithm:**

**Check Base Case**: If the array has one or no elements, return the array.

**Divide**: Find the middle index and split the array into two halves:

* Left half: arr[left:mid]
* Right half: arr[mid:right]

**Conquer**: Recursively call Merge Sort on both halves.

**Merge**:

* Create temporary arrays for left and right.
* Initialize pointers for left, right, and the main array.
* Compare elements from both arrays and place the smaller one into the main array.
* If elements remain in one half, append them to the main array.

**Return the Merged Array**.

**Code**:

#include <stdio.h>

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

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

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

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

for (j = 0; j < n2; j++)

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

i = 0; j = 0; k = left;

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

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

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

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void printArray(int arr[], int size) {

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

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

printf("\n");

}

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

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

printf("Unsorted array: \n");

printArray(arr, arrSize);

mergeSort(arr, 0, arrSize - 1);

printf("Sorted array: \n");

printArray(arr, arrSize);

return 0;

}

**Output:**

****

**Program-5 WAP to implement Quick Sort:**

**Theory:**

Quick Sort also employs a divide-and-conquer strategy but operates differently by selecting a "pivot" element. The array is partitioned into elements less than the pivot and elements greater than the pivot. This process is performed recursively on the sub-arrays until they are sorted.

Characteristics**:**

* **In-Place:** Sorts the array without needing extra space for a copy.
* **Time Complexity:** O(n log n) on average, but O(n²) in the worst case (when the smallest or largest element is consistently chosen as the pivot).
* **Not Stable:** The relative order of equal elements may not be preserved.

**Complexity:**

* Time: O(n log n) on average, O(n²) in the worst case
* Space: O(log n) (for recursion stack)

**Algorithm**:

* Choose a Pivot: Select the last element as the pivot.

Partition:

* Initialize a pointer i to keep track of the smaller elements.
* Iterate through the array with a pointer j:

If arr[j] < pivot, increment i and swap arr[i] with arr[j].

* After the loop, swap the pivot with arr[i + 1] to place it in the correct position.

Recursively Apply:

* Call Quick Sort on the left sub-array (low to pi - 1).
* Call Quick Sort on the right sub-array (pi + 1 to high).

Return the Sorted Array.

**Code**:

#include <stdio.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 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);

}

}

void printArray(int arr[], int size) {

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

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

printf("\n");

}

int main() {

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

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

printf("Unsorted array: \n");

printArray(arr, arrSize);

quickSort(arr, 0, arrSize - 1);

printf("Sorted array: \n");

printArray(arr, arrSize);

return 0;

}

**Output**:



**Program-6: WAP to implement Heap Sort**

**Theory:**

Heap Sort utilizes a binary heap data structure to sort elements. It first transforms the array into a max heap (where each parent node is greater than or equal to its child nodes), then repeatedly extracts the maximum element (the root of the heap) and rebuilds the heap until all elements are sorted.

Characteristics:

* In-Place: Requires only a constant amount of additional storage space.
* Time Complexity: O(n log n) for all cases.
* Not Stable: The order of equal elements can change

**Complexity**:

Time: O(n log n) for all cases

Space: O(1) (in-place sorting)

**Algorithm**:

* Build a Max Heap:
* For each non-leaf node from the bottom up, apply the heapify process:
  + Compare the node with its children.
  + Swap with the largest child if necessary and continue heapifying.
* Extract Elements:
* Swap the root (largest element) with the last element in the heap.
* Reduce the heap size by one.
* Restore the Max Heap:
* Apply the heapify process on the root to maintain the heap property.
* Repeat: Continue extracting and heapifying until the heap is empty.
* Return the Sorted Array.

**Code**:

#include <stdio.h>

void swap(int\* a, int\* b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

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) {

swap(&arr[i], &arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) {

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

heapify(arr, n, i);

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

swap(&arr[0], &arr[i]);

heapify(arr, i, 0);

}

}

void printArray(int arr[], int size) {

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

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

printf("\n");

}

int main() {

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

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

printf("Unsorted array: \n");

printArray(arr, arrSize);

heapSort(arr, arrSize);

printf("Sorted array: \n");

printArray(arr, arrSize);

return 0;

}

**Output**:

