# Sorting Algorithms in C

Define important Sorting types using C language programs & Proper defination and explaination.

Sorting algorithms are fundamental in computer science for organizing data efficiently. Here's a brief overview of some important sorting algorithms along with their C language implementations and explanations.

## 1. Bubble Sort

**Definition:** Bubble Sort is a simple comparison-based algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

**C Program:**

c

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (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;
            }
        }
    }
}

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

**Explanation:** In Bubble Sort, larger elements "bubble" to the top of the array. The algorithm has a time complexity of $O(n^2)$, making it inefficient on large lists.

## 2. Selection Sort

**Definition:** Selection Sort works by dividing the list into two parts: the sorted part and the unsorted part. It repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted part and moves it to the end of the sorted part.
**C Program:**

c

```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

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

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

**Explanation:** Selection Sort has a time complexity of $O(n^2)$ and is efficient for small lists. It works well for educational purposes and understanding basic sorting principles.

## 3. Insertion Sort

**Definition:** Insertion Sort builds the final sorted array one item at a time. It picks elements from the unsorted part and inserts them into their correct position in the sorted part.
**C Program:**

c

```c
#include <stdio.h>

void insertionSort(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;
        }
    }

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

    int main() {
        int arr[] = {64, 34, 25, 12, 22};
        int n = sizeof(arr)/sizeof(arr[0]);
        insertionSort(arr, n);
        printf("Sorted array: \n");
        printArray(arr, n);
        return 0;
    }
```

**Explanation:** Insertion Sort has a time complexity of $O(n^2)$ but is efficient for small or partially sorted arrays. It is adaptive and can be efficient for data that is already nearly sorted.

## 4. Merge Sort

**Definition:** Merge Sort is a divide-and-conquer algorithm. It divides the array into halves, recursively sorts each half, and then merges the sorted halves to produce the sorted array.
**C Program:**

c

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

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    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++;
    }

    free(L);
    free(R);
}

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);
    }
}

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

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, arr_size - 1);
    printf("Sorted array: \n");
    printArray(arr, arr_size);
    return 0;
}
```

**Explanation:** Merge Sort has a time complexity of $O(n \log n)$ and is more efficient for large lists. It is a stable sort and performs well on large datasets.

## 5. Quick Sort

**Definition:** Quick Sort is a divide-and-conquer algorithm that picks an element as a pivot and partitions the array around the pivot. The sub-arrays are then sorted recursively.
**C Program:**

c

```
#include <stdio.h>

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    int 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);
    }
}

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

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: \n");
    printArray(arr, n);
    return 0;
}
```

**Explanation:** Quick Sort has a time complexity of $O(n \log n)$ on average but can degrade to $O(n^2)$ in the worst case. It is efficient for large lists and is widely used in practice.

## Summary

- **Bubble Sort:** Simple, $O(n^2)$ complexity, good for small lists.
- **Selection Sort:** Simple, $O(n^2)$ complexity, intuitive but not efficient for large lists.
- **Insertion Sort:** $O(n^2)$ complexity, efficient for small or nearly sorted lists.
- **Merge Sort:** $O(n \log n)$ complexity, stable and efficient for large datasets.
- **Quick Sort:** $O(n \log n)$ average complexity, efficient but can degrade to $O(n^2)$ in the worst case.

Each sorting algorithm has its advantages and is suitable for different scenarios based on the data size and specific requirements.