SCHOOL OF COMPUTING

Vishal M.D.

CH.SC.U4CSE2415

0

Week – 3

Date - 03/01/2026

Design and Analysis of Algorithm(23CSE211)

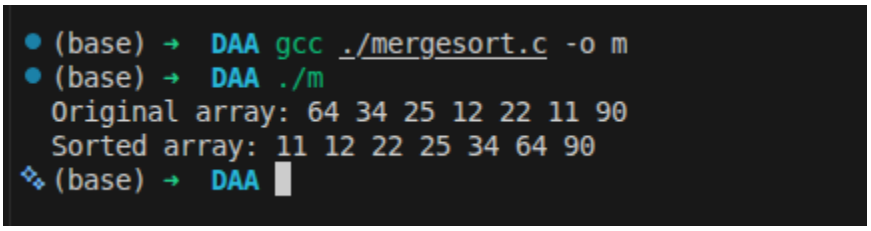
1. Merge Sort

Code:

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

void merge(int arr[], int left, int mid, int right) {
int i = left, j = mid + 1, k = 0;
int *temp = (int *)malloc((right - left + 1) * sizeof(int));
while (i <= mid && j <= right) {
if (arr[i] <= arr[j]) {
temp[k++] = arr[i++];
} else {
temp[k++] = arr[j++];
}
}
while (i <= mid) {
temp[k++] = arr[i++];
}
while (j <= right) {
temp[k++] = arr[j++];
}
for (i = left, k = 0; i <= right; i++, k++) {
arr[i] = temp[k];
}
free(temp);
}

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 n) {
for (int i = 0; i < n; 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]);
printf("Original array: ");
printArray(arr, n);
mergeSort(arr, 0, n - 1);
printf("Sorted array: ");
```

```
printArray(arr, n);
return 0;
}
```

Output:



Space Complexity: O(n log n)
Justification: The algorithm recursively divides the array into halves, which takes log n levels of division. At each level, it performs a linear merge process requiring n operations, resulting in a consistent n x log n performance.

Time Complexity: O(n)
Justification: Merge Sort requires an auxiliary array to temporarily hold elements during the merge process. This additional memory is proportional to the size of the input array.

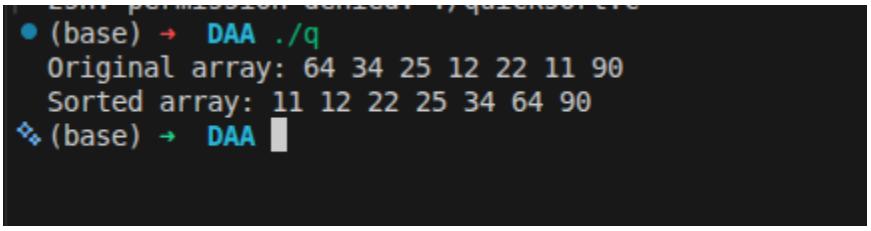
2. Quick Sort

Code:

```c
#include <stdio.h>

void swap(int *a, int *b)
{
int temp = *a;
*a = *b;
*b = temp;
}

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

for (int j = low; j < high; j++)
{
if (arr[j] < pivot)
{
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]);
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 n)
{
for (int i = 0; i < n; 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]);
```

```
printf("Original array: ");
printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array: ");
printArray(arr, n);

return 0;
}
```

Output:



Space Complexity: $O(n^2)$
Justification: This occurs when the pivot consistently picks the smallest or largest element (e.g., on a sorted array). This results in highly unbalanced partitions where one side has 0 elements and the other has n-1, leading to n recursive levels with O(n) work each.

Time Complexity: O(n)
Justification: In the worst-case scenario of an unbalanced partition, the recursion stack depth increases from the ideal log n to n. Each recursive call stays on the stack until it finishes, requiring memory proportional to the number of elements.