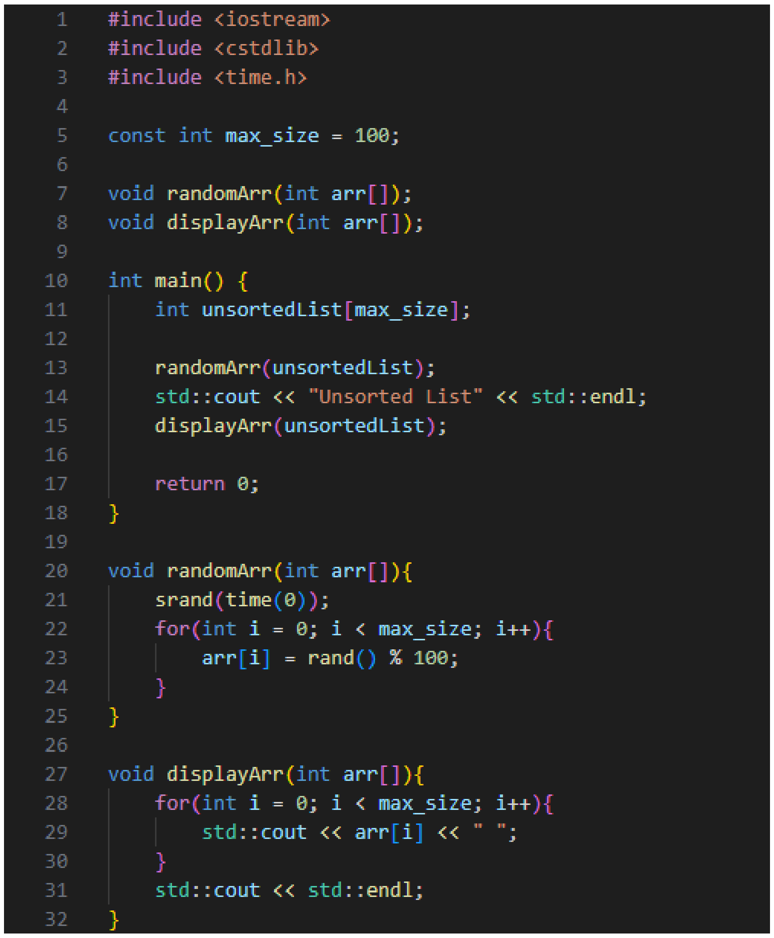
Hands-on Activity 8.1	
Sorting Algorithms Pt2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09/27/2025
Section: CPE21S4	**Date Submitted:** 09/27/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

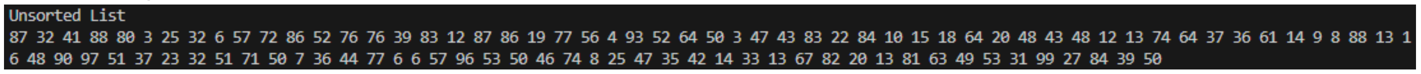
6. Output

Table 8.1 Code



```cpp
#include <iostream>
#include <cstdlib>
#include <time.h>

const int max_size = 100;

void randomArr(int arr[]);
void displayArr(int arr[]);

int main() {
    int unsortedList[max_size];

    randomArr(unsortedList);
    std::cout << "Unsorted List" << std::endl;
    displayArr(unsortedList);

    return 0;
}

void randomArr(int arr[]){
    srand(time(0));
    for(int i = 0; i < max_size; i++){
        arr[i] = rand() % 100;
    }
}

void displayArr(int arr[]){
    for(int i = 0; i < max_size; i++){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

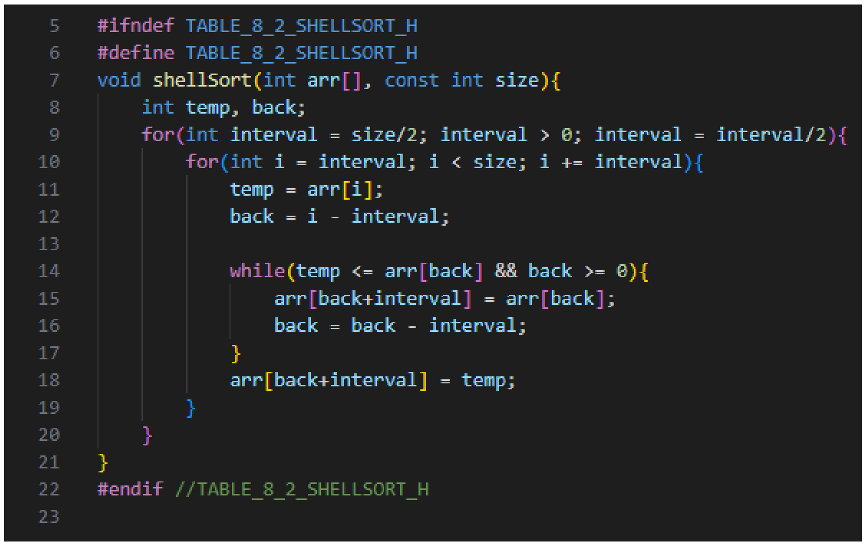
Console Output:



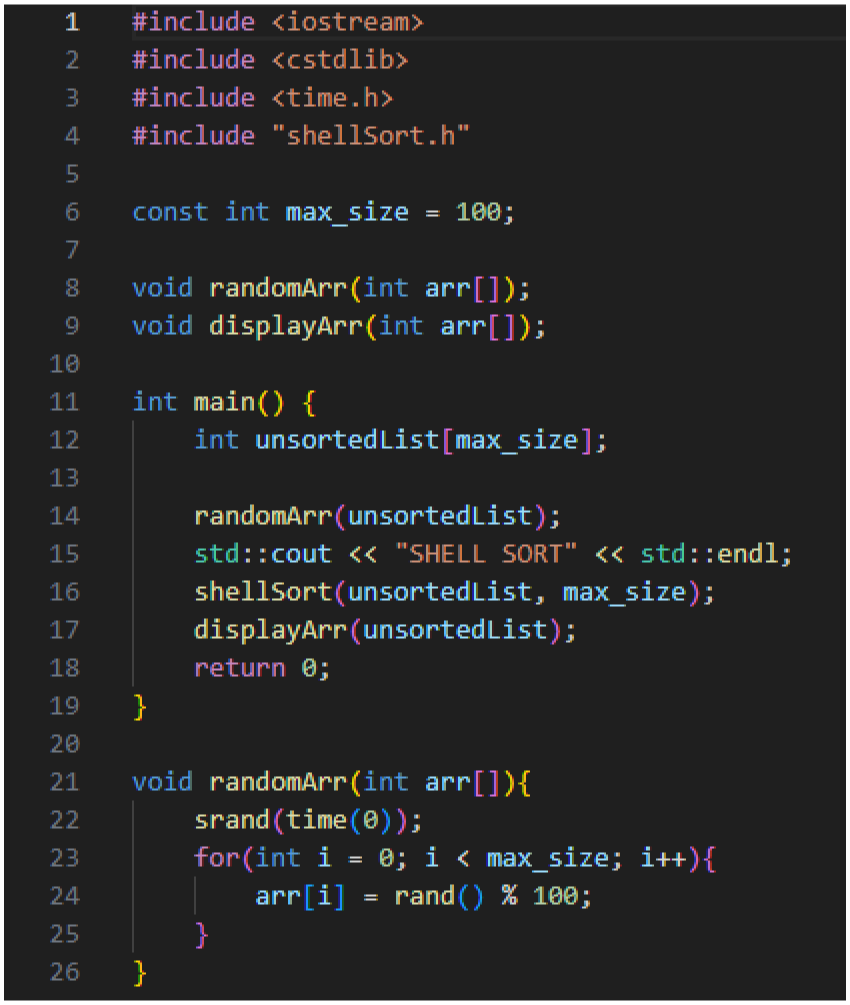
```
Unsorted List
87 32 41 88 80 3 25 32 6 57 72 86 52 76 76 39 83 12 87 86 19 77 56 4 93 52 64 50 3 47 43 83 22 84 10 15 18 64 20 48 43 48 12 13 74 64 37 36 61 14 9 8 88 13 1
6 48 90 97 51 37 23 32 51 71 50 7 36 44 77 6 6 57 96 53 50 46 74 8 25 47 35 42 14 33 13 67 82 20 13 81 63 49 53 31 99 27 84 39 50
```

Explanation:
The main() function declares an unsortedList array of size 100, immediately calling randomArr to populate it. The randomArr function first ensures the generation of distinct random sequences via srand(time(0)) and then fills the array elements within a for loop, using rand() \% 100 to limit values to the 0−99 range. After generation, main() prints the "Unsorted List" title and then calls displayArr, which uses a final for loop to iterate through the array and print each element to the console, concluding the setup phase for a list of 100 random integers.

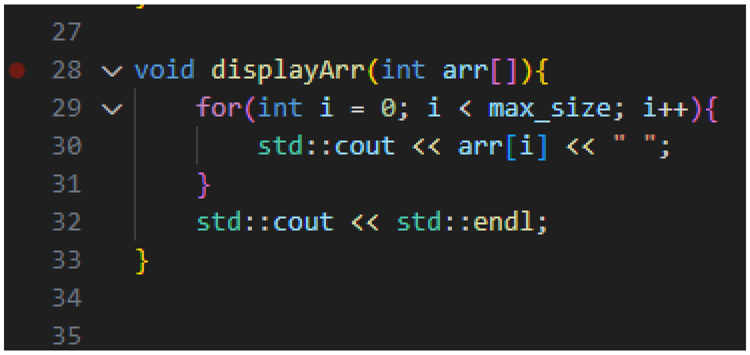
Table 8.2 Code



```cpp
#ifndef TABLE_8_2_SHELLSORT_H
#define TABLE_8_2_SHELLSORT_H
void shellSort(int arr[], const int size){
    int temp, back;
    for(int interval = size/2; interval > 0; interval = interval/2){
        for(int i = interval; i < size; i += interval){
            temp = arr[i];
            back = i - interval;

            while(temp <= arr[back] && back >= 0){
                arr[back+interval] = arr[back];
                back = back - interval;
            }
            arr[back+interval] = temp;
        }
    }
}
#endif //TABLE_8_2_SHELLSORT_H

```

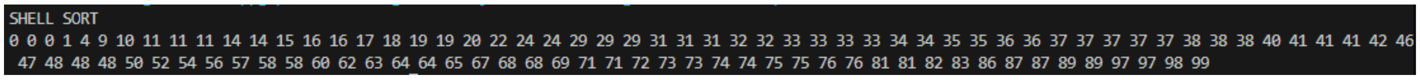


```cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "shellSort.h"

const int max_size = 100;

void randomArr(int arr[]);
void displayArr(int arr[]);

int main() {
    int unsortedList[max_size];

    randomArr(unsortedList);
    std::cout << "SHELL SORT" << std::endl;
    shellSort(unsortedList, max_size);
    displayArr(unsortedList);
    return 0;
}

void randomArr(int arr[]){
    srand(time(0));
    for(int i = 0; i < max_size; i++){
        arr[i] = rand() % 100;
    }
}
```



```cpp
void displayArr(int arr[]){
    for(int i = 0; i < max_size; i++){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}


```

Console Output:



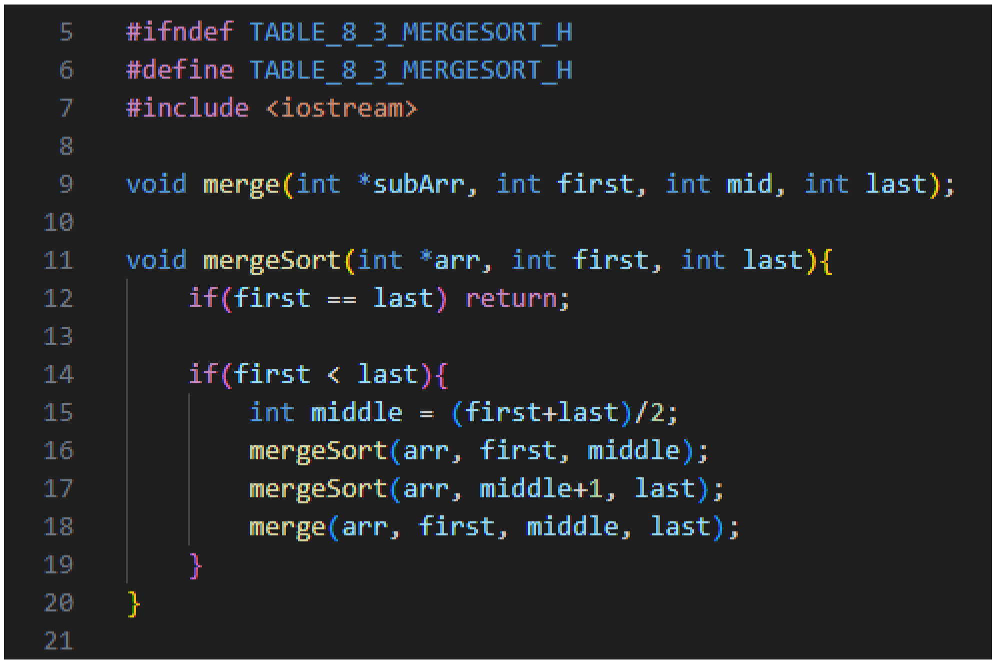
```
SHELL SORT
0 0 0 1 4 9 10 11 11 11 14 14 15 16 16 17 18 19 19 20 22 24 24 29 29 29 31 31 31 32 32 33 33 33 33 34 34 35 35 36 36 37 37 37 37 37 38 38 38 40 41 41 41 42 46
 47 48 48 48 50 52 54 56 57 58 58 60 62 63 64 64 65 67 68 68 69 71 71 72 73 73 74 74 75 75 76 76 81 81 82 83 86 87 87 89 89 97 97 98 99
```

Explanation:

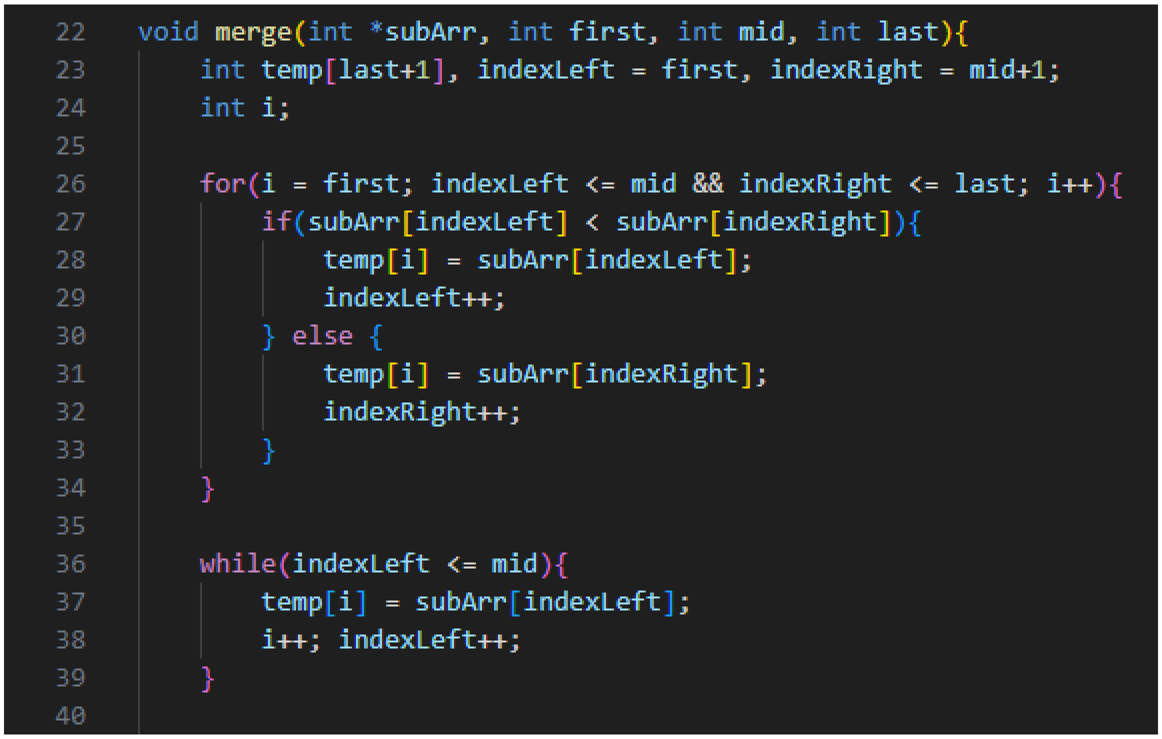
The Shell Sort operates by breaking the array into smaller sub-lists based on a continuously decreasing interval (gap). The function begins with an interval equal to half the array size, performing a gapped insertion sort on elements separated by this interval. This process allows elements far apart to be quickly compared and swapped, significantly reducing the number of moves needed later. The while loop within the inner for loop handles this insertion: an element (temp) is held, and comparisons are made across the gap (arr[back] && back >= 0), shifting elements until the correct position is found. The interval is then halved (interval = interval/2) until the gap is 1, at which point the algorithm performs a final, efficient standard insertion sort on the nearly sorted array, ensuring all elements are in their correct positions.

The main function is like the previous program as it relies on three standard headers, defines a fixed max_size of 100, and utilizes the randomArr function which similarly seeds the generator with srand(time(0)) to populate the array with values from 0 to 99. The main() function's flow is direct: it calls randomArr to populate the unsortedList, prints the "SHELL SORT" title, then executes the core shellSort logic from the linked header file, and concludes by calling displayArr to confirm the final, sorted state of the array.

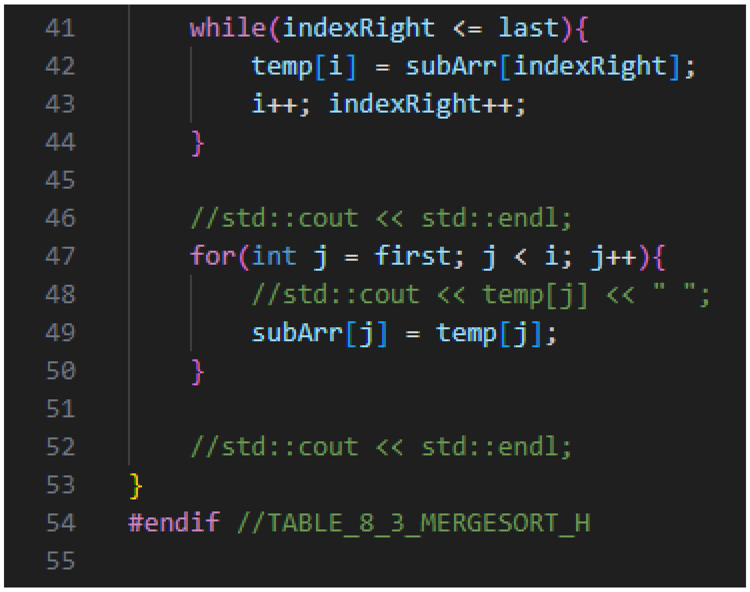
Table 8.3 Code



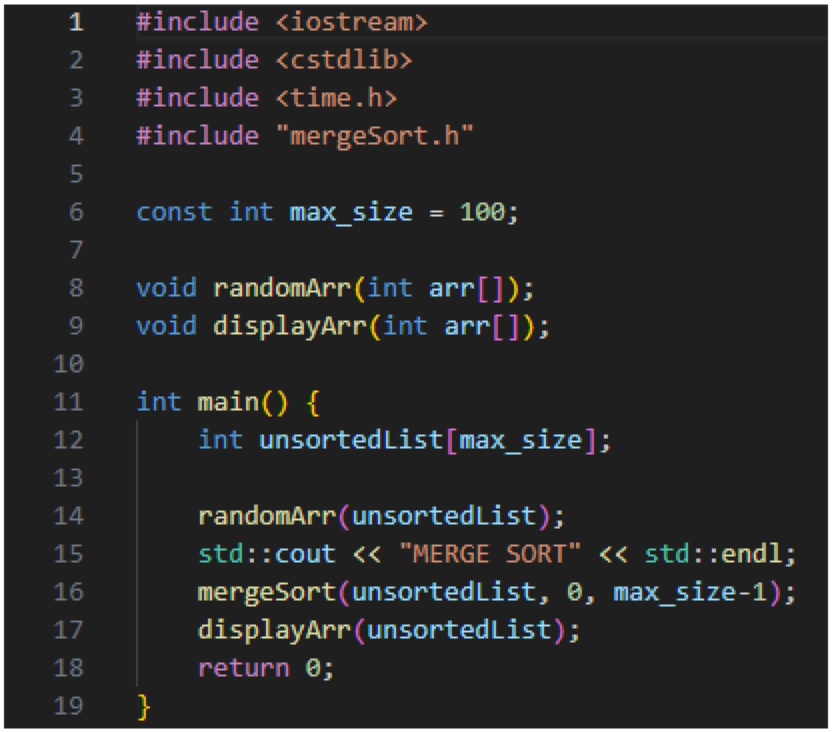
```cpp
#ifndef TABLE_8_3_MERGESORT_H
#define TABLE_8_3_MERGESORT_H
#include <iostream>

void merge(int *subArr, int first, int mid, int last);

void mergeSort(int *arr, int first, int last){
    if(first == last) return;

    if(first < last){
        int middle = (first+last)/2;
        mergeSort(arr, first, middle);
        mergeSort(arr, middle+1, last);
        merge(arr, first, middle, last);
    }
}

```



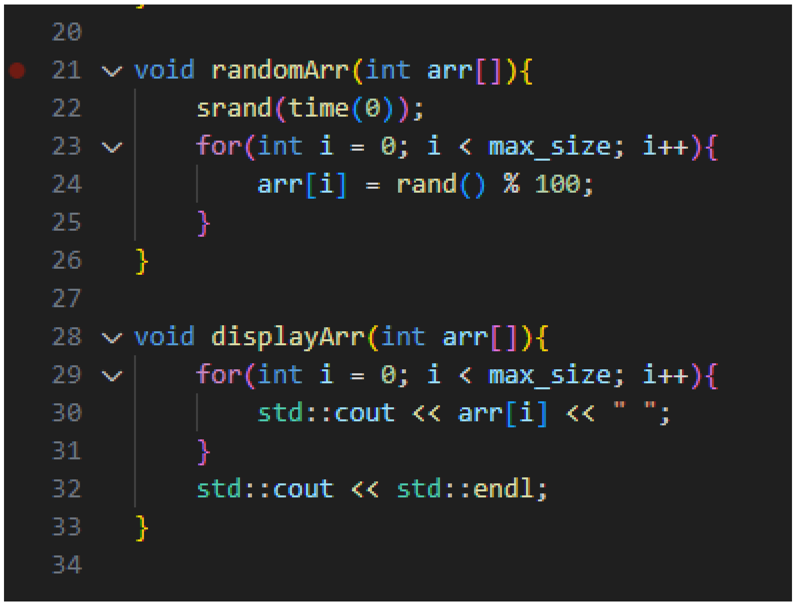
```cpp
void merge(int *subArr, int first, int mid, int last){
    int temp[last+1], indexLeft = first, indexRight = mid+1;
    int i;

    for(i = first; indexLeft <= mid && indexRight <= last; i++){
        if(subArr[indexLeft] < subArr[indexRight]){
            temp[i] = subArr[indexLeft];
            indexLeft++;
        } else {
            temp[i] = subArr[indexRight];
            indexRight++;
        }
    }

    while(indexLeft <= mid){
        temp[i] = subArr[indexLeft];
        i++; indexLeft++;
    }

```



```cpp
    while(indexRight <= last){
        temp[i] = subArr[indexRight];
        i++; indexRight++;
    }

    //std::cout << std::endl;
    for(int j = first; j < i; j++){
        //std::cout << temp[j] << " ";
        subArr[j] = temp[j];
    }

    //std::cout << std::endl;
}
#endif //TABLE_8_3_MERGESORT_H

```

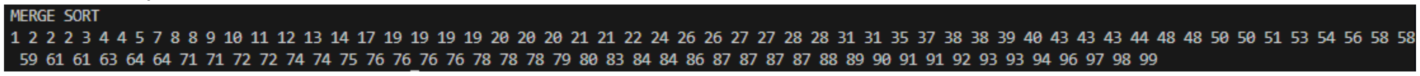


```cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "mergeSort.h"

const int max_size = 100;

void randomArr(int arr[]);
void displayArr(int arr[]);

int main() {
    int unsortedList[max_size];

    randomArr(unsortedList);
    std::cout << "MERGE SORT" << std::endl;
    mergeSort(unsortedList, 0, max_size-1);
    displayArr(unsortedList);
    return 0;
}
```



```cpp

void randomArr(int arr[]){
    srand(time(0));
    for(int i = 0; i < max_size; i++){
        arr[i] = rand() % 100;
    }
}

void displayArr(int arr[]){
    for(int i = 0; i < max_size; i++){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

```

Console Output:



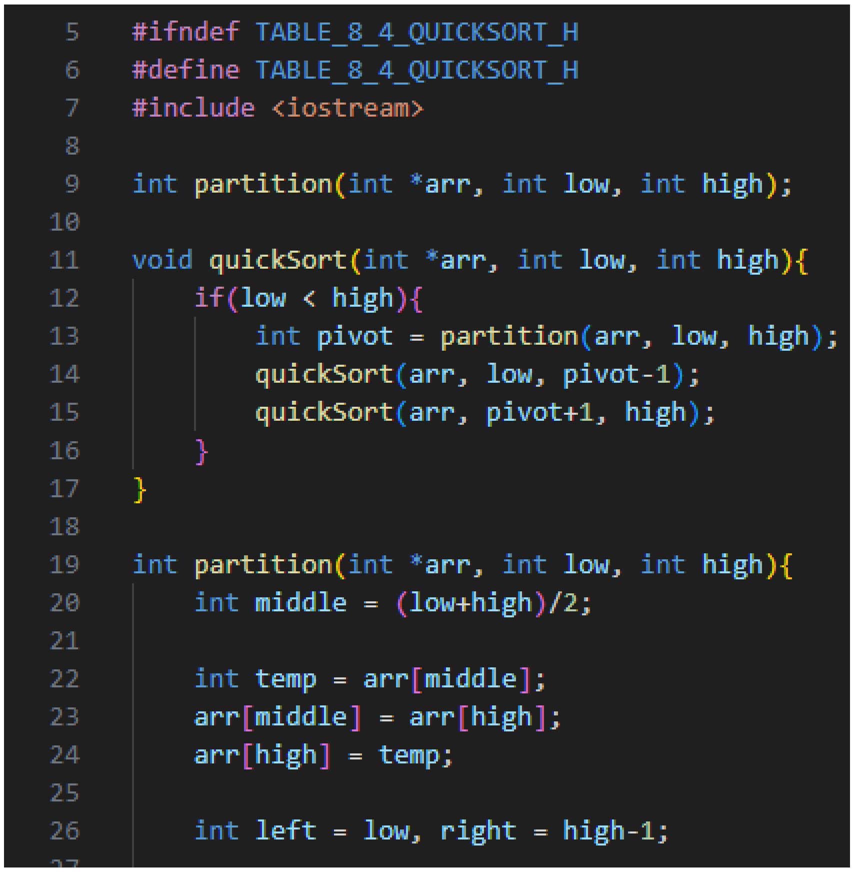
```
MERGE SORT
1 2 2 2 3 4 4 5 7 8 8 9 10 11 12 13 14 17 19 19 19 19 20 20 20 21 21 22 24 26 26 27 27 28 28 31 31 35 37 38 38 39 40 43 43 43 44 48 48 50 50 51 53 54 56 58 58
 59 61 61 63 64 64 71 71 72 72 74 74 75 76 76 76 76 78 78 78 79 80 83 84 84 86 87 87 87 87 88 89 90 91 91 92 93 93 94 96 97 98 99
```

Explanation:

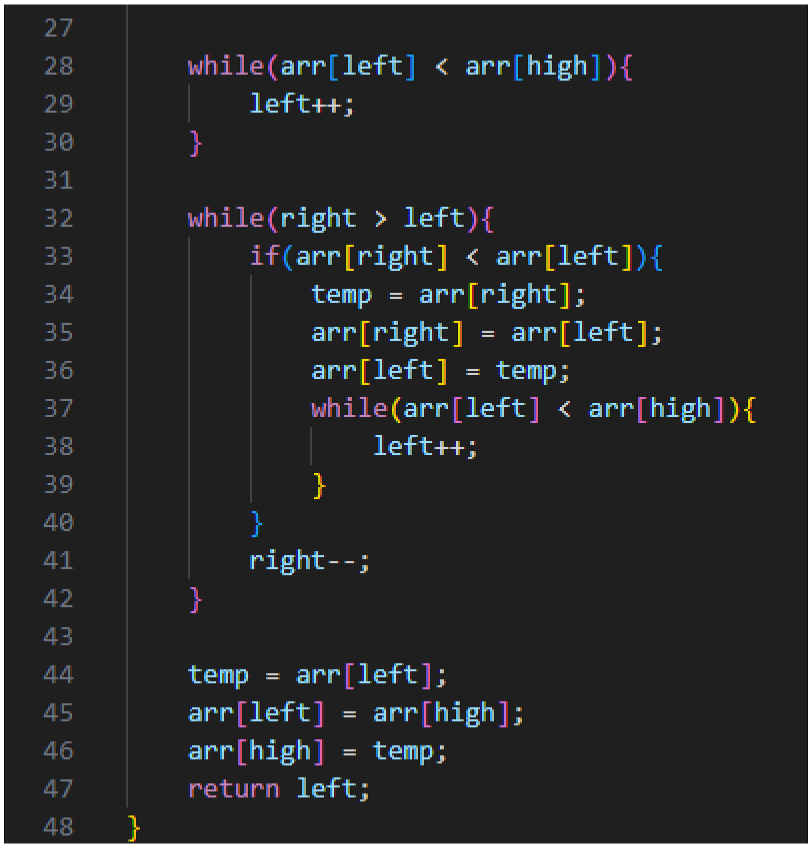
The mergeSort process starts by checking the base case: if (first < last); if the sub-array has only one element, the function simply returns. If not, it finds the middle index and recursively calls itself to split the array into two halves: mergeSort(arr, first, middle) and mergeSort(arr, middle + 1, last). This recursive division continues until the entire array is broken down into individual elements (sub-arrays of size one). Once the division is complete, the function calls the merge function to begin the combining and sorting phase.

The merge function is where the actual sorting and combining happen. It takes two adjacent sorted sub-arrays (from first to mid and from mid+1 to last) and merges them into a single sorted array. It uses a temporary array, temp, and three index pointers: IndexLeft, IndexRight, and i (for the temp array). The main for loop compares elements from the two sub-arrays: if (subArr[IndexLeft] <= subArr[IndexRight]), the smaller element is moved to the temp array, and its corresponding index is incremented. After one of the sub-arrays is exhausted, the two subsequent while loops efficiently copy any remaining elements from the unexhausted sub-array into temp. Finally, a loop copies the complete, sorted range back from the temp array into the original subArr, successfully combining the two halves.

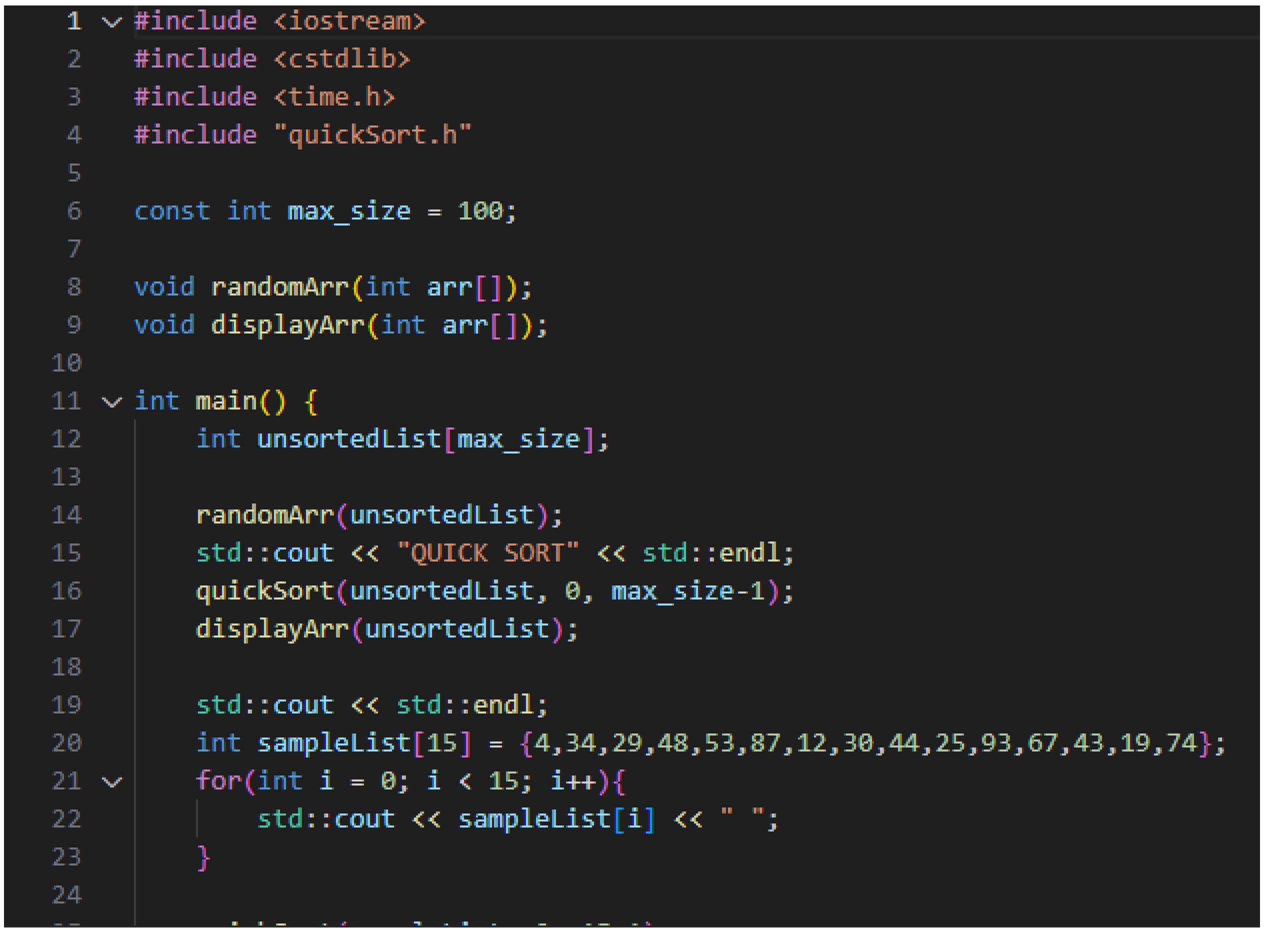
Table 8.4 Code:



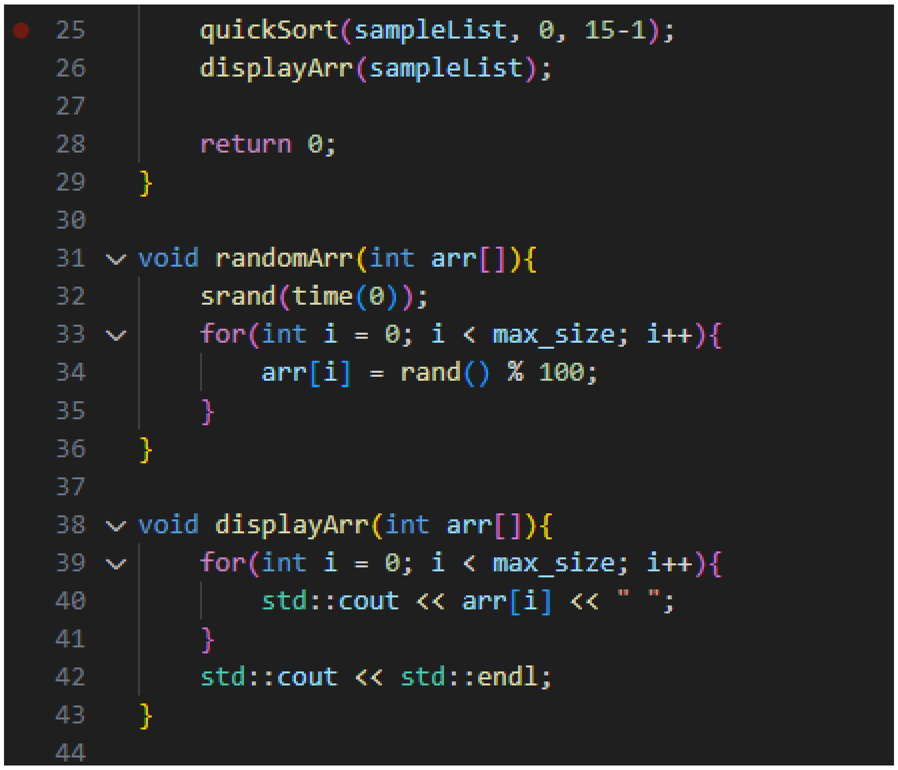
```cpp
#ifndef TABLE_8_4_QUICKSORT_H
#define TABLE_8_4_QUICKSORT_H
#include <iostream>

int partition(int *arr, int low, int high);

void quickSort(int *arr, int low, int high){
    if(low < high){
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot-1);
        quickSort(arr, pivot+1, high);
    }
}

int partition(int *arr, int low, int high){
    int middle = (low+high)/2;

    int temp = arr[middle];
    arr[middle] = arr[high];
    arr[high] = temp;

    int left = low, right = high-1;
```



```cpp

    while(arr[left] < arr[high]){
        left++;
    }

    while(right > left){
        if(arr[right] < arr[left]){
            temp = arr[right];
            arr[right] = arr[left];
            arr[left] = temp;
            while(arr[left] < arr[high]){
                left++;
            }
        }
        right--;
    }

    temp = arr[left];
    arr[left] = arr[high];
    arr[high] = temp;
    return left;
}
```

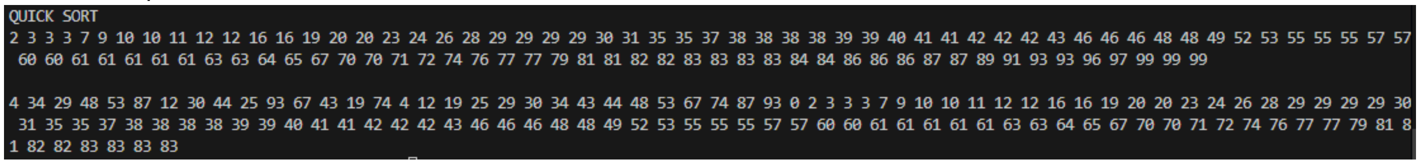


```cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "quickSort.h"

const int max_size = 100;

void randomArr(int arr[]);
void displayArr(int arr[]);

int main() {
    int unsortedList[max_size];

    randomArr(unsortedList);
    std::cout << "QUICK SORT" << std::endl;
    quickSort(unsortedList, 0, max_size-1);
    displayArr(unsortedList);

    std::cout << std::endl;
    int sampleList[15] = {4,34,29,48,53,87,12,30,44,25,93,67,43,19,74};
    for(int i = 0; i < 15; i++){
        std::cout << sampleList[i] << " ";
    }

```



```cpp
    quickSort(sampleList, 0, 15-1);
    displayArr(sampleList);

    return 0;
}

void randomArr(int arr[]){
    srand(time(0));
    for(int i = 0; i < max_size; i++){
        arr[i] = rand() % 100;
    }
}

void displayArr(int arr[]){
    for(int i = 0; i < max_size; i++){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

```

Console Output:



```
QUICK SORT
2 3 3 3 7 9 10 10 11 12 12 16 16 19 20 20 23 24 26 28 29 29 29 29 30 31 35 35 37 38 38 38 38 39 39 40 41 41 42 42 42 43 46 46 46 48 48 49 52 53 55 55 55 57 57
 60 60 61 61 61 61 61 63 63 64 65 67 70 70 71 72 74 76 77 77 79 81 81 82 82 83 83 83 83 84 84 86 86 86 87 87 89 91 93 93 96 97 99 99 99

4 34 29 48 53 87 12 30 44 25 93 67 43 19 74 4 12 19 25 29 30 34 43 44 48 53 67 74 87 93 0 2 3 3 3 7 9 10 10 11 12 12 16 16 19 20 20 23 24 26 28 29 29 29 29 30
 31 35 35 37 38 38 38 38 39 39 40 41 41 42 42 42 43 46 46 46 48 48 49 52 53 55 55 55 57 57 60 60 61 61 61 61 61 63 63 64 65 67 70 70 71 72 74 76 77 77 79 81 8
1 82 82 83 83 83 83
```

Explanation:
The quickSort function embodies the Divide and Conquer paradigm, similar to Merge Sort, but it performs partitioning in place. It operates recursively, first checking the condition if (low < high) to ensure the sub-array has more than one element. If it does, it calls partition to select and position a pivot element, returning the pivot's final index. The algorithm then recursively calls itself twice on the two resulting sub-arrays: quickSort(arr, low, pivot - 1) for all elements smaller than the pivot, and quickSort(arr, pivot + 1, high) for all elements larger than the pivot. This process continues until all sub-arrays are sorted, achieving the final sorted list.

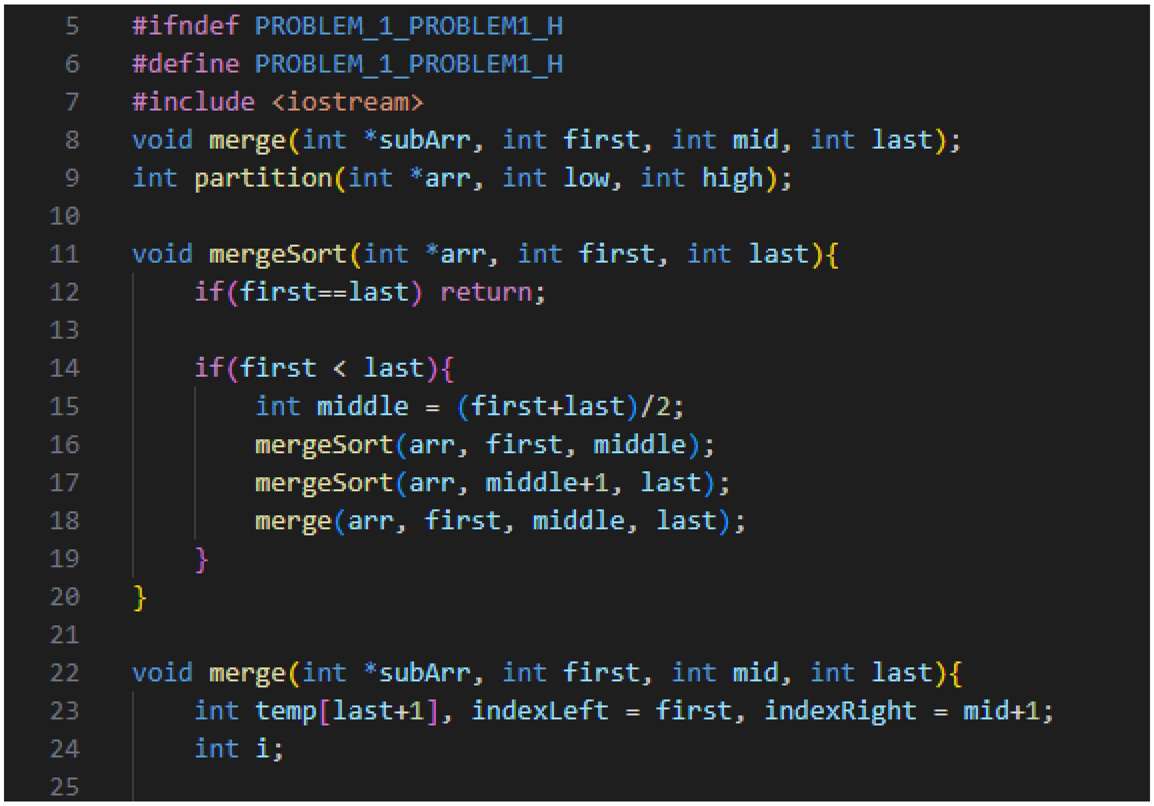
The partition function is the core of Quick Sort, responsible for selecting a pivot and arranging the sub-array so that all elements less than the pivot come before it, and all greater elements come after it. In this implementation, the pivot is selected as the element at the middle index ((low+high)/2). This pivot is then swapped with the element at the high index for simplification. The function uses two pointers, left and right, to traverse the array, comparing elements against the pivot's value (stored temporarily at arr[high]) and swapping them to ensure the partition rule is met. The process is concluded by swapping the pivot element (currently at arr[high]) with the element at the final left index, placing the pivot in its correct sorted position and returning that index.

7. Supplementary Activity

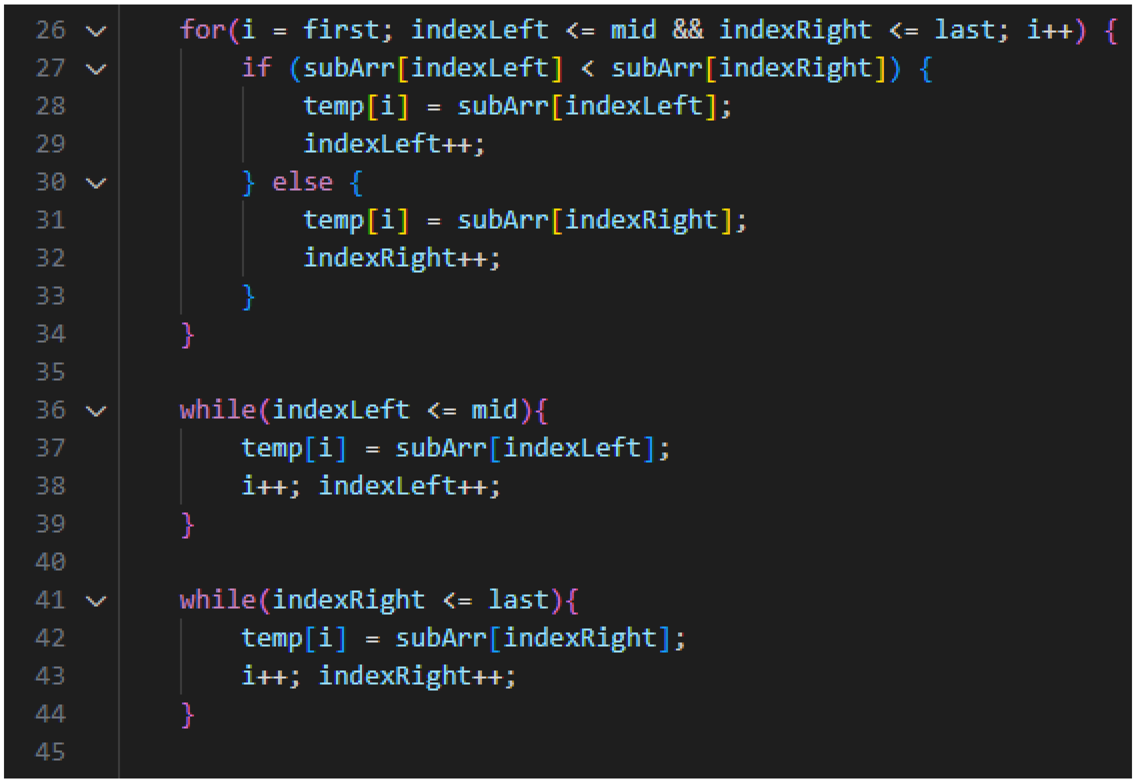
Problem 1: Can we sort the left sub list and right sub list from the partition method in quick sort using other sorting algorithms? Demonstrate an example.

A Hybrid Sorting Algorithm is the standard approach for this optimization, combining Quick Sort's efficient recursive partitioning with a different algorithm to sort the resulting sub-arrays, most commonly Insertion Sort. Since Quick Sort's recursive overhead on very small sub-lists is inefficient, the algorithm switches to the low-overhead Insertion Sort once a sub-list size falls below a specific threshold.

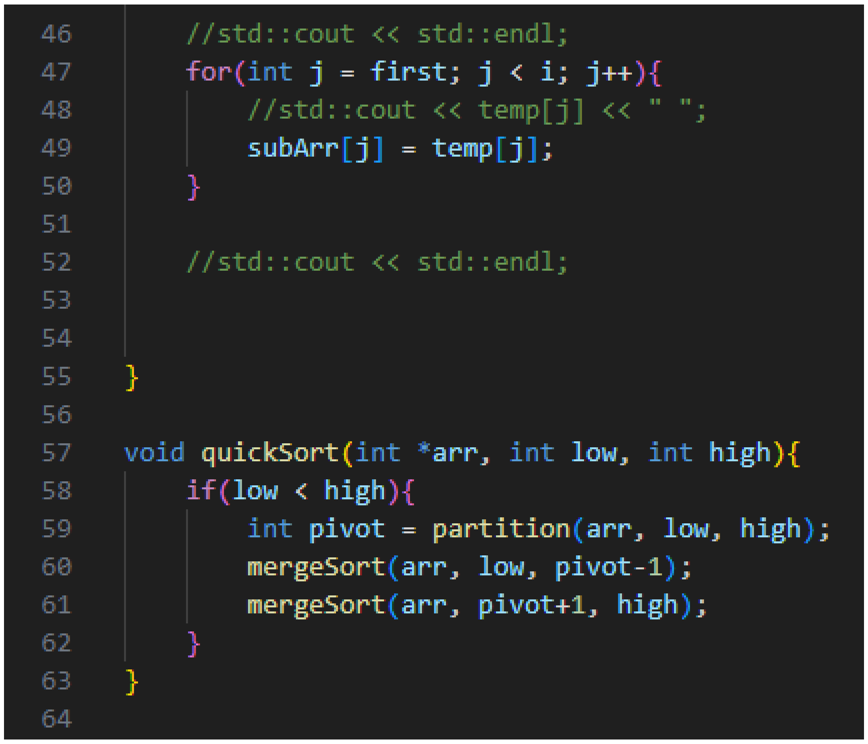
Problem 1 Code:



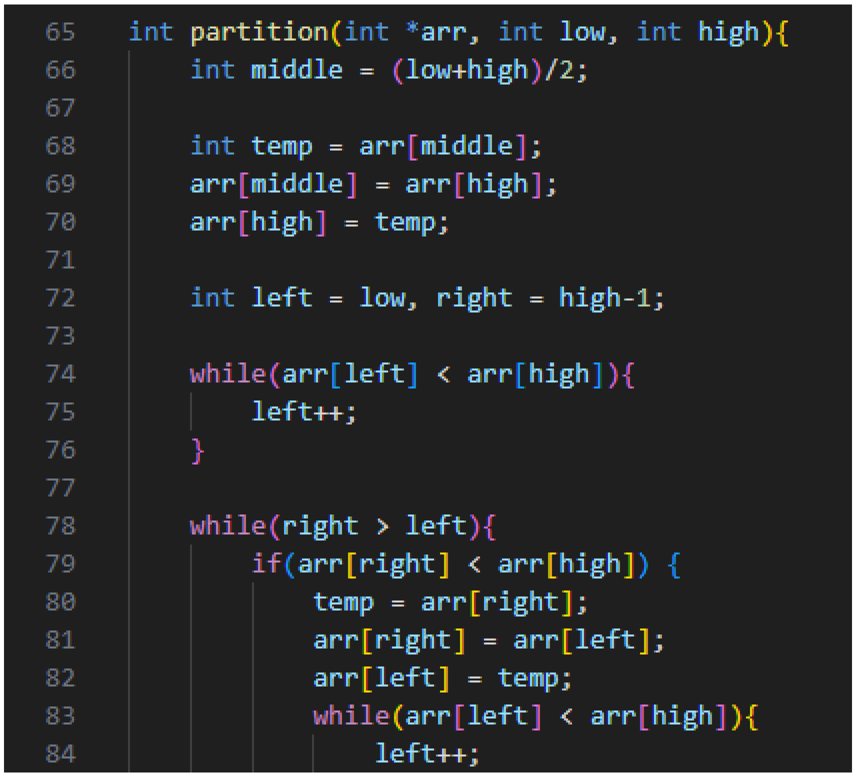
```
#ifndef PROBLEM_1_PROBLEM1_H
#define PROBLEM_1_PROBLEM1_H
#include <iostream>
void merge(int *subArr, int first, int mid, int last);
int partition(int *arr, int low, int high);

void mergeSort(int *arr, int first, int last){
    if(first==last) return;

    if(first < last){
        int middle = (first+last)/2;
        mergeSort(arr, first, middle);
        mergeSort(arr, middle+1, last);
        merge(arr, first, middle, last);
    }
}

void merge(int *subArr, int first, int mid, int last){
    int temp[last+1], indexLeft = first, indexRight = mid+1;
    int i;

```



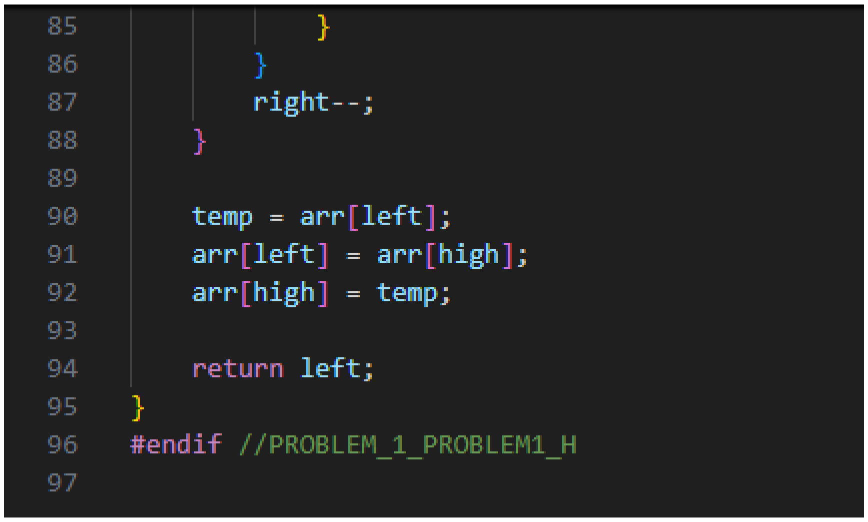
```cpp
    for(i = first; indexLeft <= mid && indexRight <= last; i++) {
        if (subArr[indexLeft] < subArr[indexRight]) {
            temp[i] = subArr[indexLeft];
            indexLeft++;
        } else {
            temp[i] = subArr[indexRight];
            indexRight++;
        }
    }

    while(indexLeft <= mid){
        temp[i] = subArr[indexLeft];
        i++; indexLeft++;
    }

    while(indexRight <= last){
        temp[i] = subArr[indexRight];
        i++; indexRight++;
    }

```



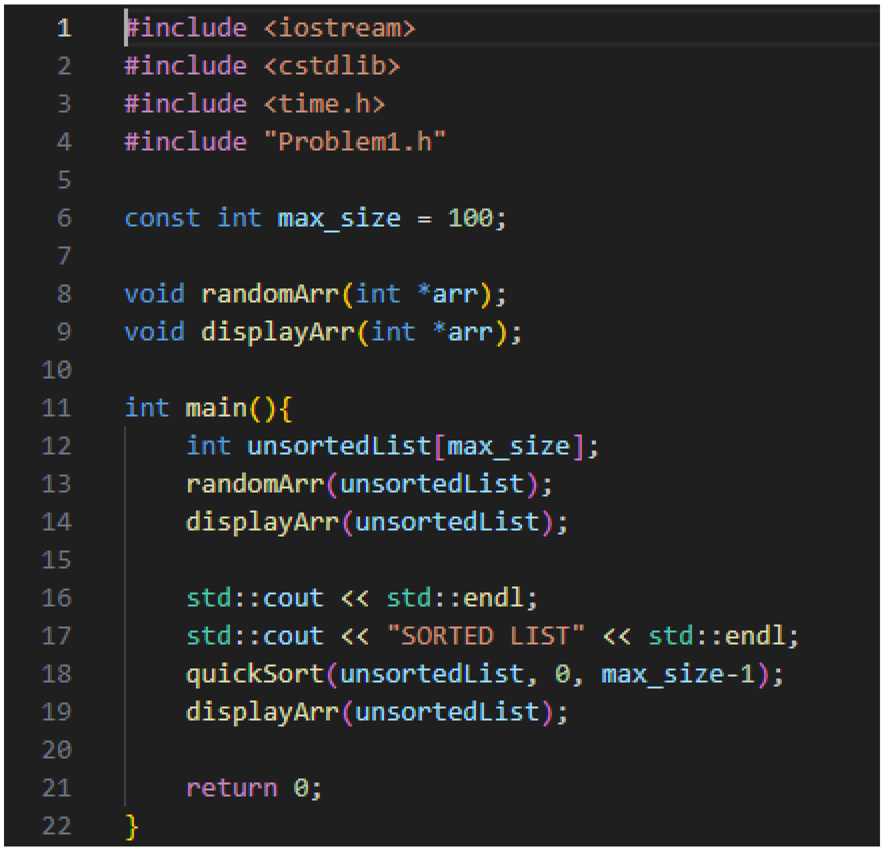
```cpp
    //std::cout << std::endl;
    for(int j = first; j < i; j++){
        //std::cout << temp[j] << " ";
        subArr[j] = temp[j];
    }

    //std::cout << std::endl;


}

void quickSort(int *arr, int low, int high){
    if(low < high){
        int pivot = partition(arr, low, high);
        mergeSort(arr, low, pivot-1);
        mergeSort(arr, pivot+1, high);
    }
}

```



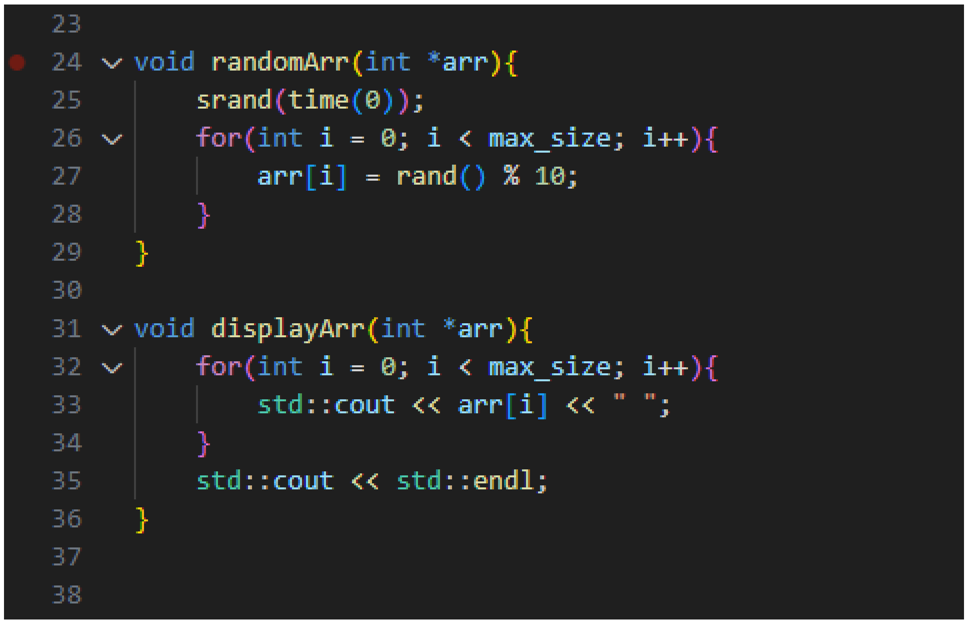
```c
int partition(int *arr, int low, int high){
    int middle = (low+high)/2;

    int temp = arr[middle];
    arr[middle] = arr[high];
    arr[high] = temp;

    int left = low, right = high-1;

    while(arr[left] < arr[high]){
        left++;
    }

    while(right > left){
        if(arr[right] < arr[high]) {
            temp = arr[right];
            arr[right] = arr[left];
            arr[left] = temp;
            while(arr[left] < arr[high]){
                left++;
```



```c
            }
        }
        right--;
    }

    temp = arr[left];
    arr[left] = arr[high];
    arr[high] = temp;

    return left;
}
#endif //PROBLEM_1_PROBLEM1_H

```

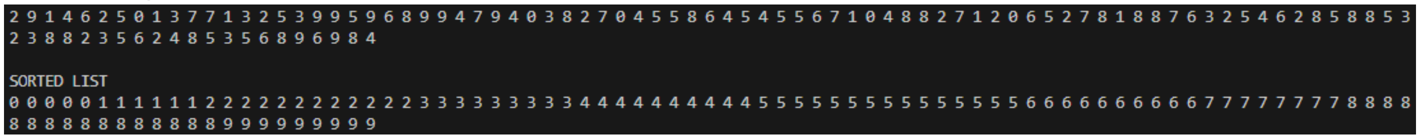


```cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "Problem1.h"

const int max_size = 100;

void randomArr(int *arr);
void displayArr(int *arr);

int main(){
    int unsortedList[max_size];
    randomArr(unsortedList);
    displayArr(unsortedList);

    std::cout << std::endl;
    std::cout << "SORTED LIST" << std::endl;
    quickSort(unsortedList, 0, max_size-1);
    displayArr(unsortedList);

    return 0;
}
```



```cpp

void randomArr(int *arr){
    srand(time(0));
    for(int i = 0; i < max_size; i++){
        arr[i] = rand() % 10;
    }
}

void displayArr(int *arr){
    for(int i = 0; i < max_size; i++){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}


```

Console Output:



```
2 9 1 4 6 2 5 0 1 3 7 7 1 3 2 5 3 9 9 5 9 6 8 9 9 4 7 9 4 0 3 8 2 7 0 4 5 5 8 6 4 5 4 5 5 6 7 1 0 4 8 8 2 7 1 2 0 6 5 2 7 8 1 8 8 7 6 3 2 5 4 6 2 8 5 8 8 5 3
2 3 8 8 2 3 5 6 2 4 8 5 3 5 6 8 9 6 9 8 4

SORTED LIST
0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 8 8 8 8
8 8 8 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9
```

Problem 2: Suppose we have an array which consists of {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74}. What sorting algorithm will give you the fastest time performance? Why can merge sort and quick sort have O(N • log N) for their time complexity?

For an array of 15 elements, the absolute fastest sorting algorithm might be Insertion Sort due to its advantage on very small lists, but for large-scale data sets, the best average-case performance comes from Quick Sort. Both Quick Sort and Merge Sort achieve an optimal time complexity of $O(N \cdot \log N)$ because they are Divide and Conquer algorithms: the $\log N$ factor represents the number of times the array must be recursively halved until the base case is reached, while the N factor represents the total amount of work (comparisons and swaps/moves) performed at each level of that recursion, ensuring that the overall workload scales linearly with the size of the array multiplied by the number of division stages.

8. Conclusion

Working on this activity showed me how tricky it can be to write C++ code to sort datasets with stuff like shell, merge, and quick sort. It pushed me to figure out how these methods work and play together in a program. I learned it's key to pick the right sorting trick for the job to keep things running smooth. This whole thing made me see how important it is to plan my code early on so it can handle different sorting tasks with versatility. It also got me into the habit of keeping my code neat and simple to avoid mistakes later.

9. Assessment Rubric